ASP.NET Core 1.0 Documentation

Microsoft

7 06, 2016

: ASP.NET 5 has been renamed to ASP.NET Core 1.0. Read more.

: 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.

Topics

1.1 Introduction to ASP.NET Core

By Daniel Roth, Rick Anderson and Shaun Luttin

ASP.NET Core is a significant redesign of ASP.NET. This topic introduces the new concepts in ASP.NET Core and explains how they help you develop modern web apps.

Sections:

- *What is ASP.NET Core?*
- *Why build ASP.NET Core?*
- *Application anatomy*
- *Startup*
- *Services*
- *Middleware*
- *Servers*
- *Content root*
- *Web root*
- *Configuration*
- *Environments*
- *Build web UI and web APIs using ASP.NET Core MVC*
- *Client-side development*
- *Next steps*

1.1.1 What is ASP.NET Core?

ASP.NET Core is a new open-source and cross-platform framework for building modern cloud based internet connected applications, such as web apps, IoT apps and mobile backends. ASP.NET Core apps can run on .NET Core or on the full .NET Framework. It was architected to provide an optimized development framework for apps that are deployed to the cloud or run on-premises. It consists of modular components with minimal overhead, so you retain flexibility while constructing your solutions. You can develop and run your ASP.NET Core apps cross-platform on Windows, Mac and Linux. ASP.NET Core is open source at GitHub.

1.1.2 Why build ASP.NET Core?

The first preview release of ASP.NET came out almost 15 years ago as part of the .NET Framework. Since then millions of developers have used it to build and run great web apps, and over the years we have added and evolved many capabilities to it.

ASP.NET Core has a number of architectural changes that result in a much leaner and modular framework. ASP.NET Core is no longer based on *System.Web.dll*. It is based on a set of granular and well factored NuGet packages. This allows you to optimize your app to include just the NuGet packages you need. The benefits of a smaller app surface area include tighter security, reduced servicing, improved performance, and decreased costs in a pay-for-what-you-use model.

With ASP.NET Core you gain the following foundational improvements:

- A unified story for building web UI and web APIs

- Integration of *modern client-side frameworks* and development workflows

- A cloud-ready environment-based *configuration system*

- Built-in *dependency injection*

- New light-weight and modular HTTP request pipeline

- Ability to host on IIS or self-host in your own process

- Built on .NET Core, which supports true side-by-side app versioning

- Ships entirely as NuGet packages

- New tooling that simplifies modern web development

- Build and run cross-platform ASP.NET apps on Windows, Mac and Linux

- Open source and community focused

1.1.3 Application anatomy

An ASP.NET Core app is simply a console app that creates a web server in its `Main` method:

```
using System;
using Microsoft.AspNetCore.Hosting;

namespace aspnetcoreapp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}
```

`Main` uses `WebHostBuilder`, which follows the builder pattern, to create a web application host. The builder has methods that define the web server (for example `UseKestrel`) and the startup class (`UseStartup`). In the example above, the Kestrel web server is used, but other web servers can be specified. We'll show more about `UseStartup`

in the next section. `WebHostBuilder` provides many optional methods including `UseIISIntegration` for hosting in IIS and IIS Express and `UseContentRoot` for specifying the root content directory. The `Build` and `Run` methods build the `IWebHost` that will host the app and start it listening for incoming HTTP requests.

1.1.4 Startup

The `UseStartup` method on `WebHostBuilder` specifies the `Startup` class for your app.

```csharp
public class Program
{
    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseStartup<Startup>()
            .Build();

        host.Run();
    }
}
```

The `Startup` class is where you define the request handling pipeline and where any services needed by the app are configured. The `Startup` class must be public and contain the following methods:

```csharp
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IApplicationBuilder app)
    {
    }
}
```

- `ConfigureServices` defines the services (see *Services* below) used by your app (such as the ASP.NET MVC Core framework, Entity Framework Core, Identity, etc.)
- `Configure` defines the *middleware* in the request pipeline
- See *Application Startup* for more details

1.1.5 Services

A service is a component that is intended for common consumption in an application. Services are made available through dependency injection. ASP.NET Core includes a simple built-in inversion of control (IoC) container that supports constructor injection by default, but can be easily replaced with your IoC container of choice. In addition to its loose coupling benefit, DI makes services available throughout your app. For example, *Logging* is available throughout your app. See *Dependency Injection* for more details.

1.1.6 Middleware

In ASP.NET Core you compose your request pipeline using *Middleware*. ASP.NET Core middleware performs asynchronous logic on an `HttpContext` and then either invokes the next middleware in the sequence or terminates the

request directly. You generally "Use" middleware by invoking a corresponding `UseXYZ` extension method on the `IApplicationBuilder` in the `Configure` method.

ASP.NET Core comes with a rich set of prebuilt middleware:

- *Static files*
- *Routing*
- *Authentication*

You can also author your own *custom middleware*.

You can use any OWIN-based middleware with ASP.NET Core. See *Open Web Interface for .NET (OWIN)* for details.

1.1.7 Servers

The ASP.NET Core hosting model does not directly listen for requests; rather it relies on an HTTP *server* implementation to forward the request to the application. The forwarded request is wrapped as a set of feature interfaces that the application then composes into an `HttpContext`. ASP.NET Core includes a managed cross-platform web server, called *Kestrel*, that you would typically run behind a production web server like IIS or nginx.

1.1.8 Content root

The content root is the base path to any content used by the app, such as its views and web content. By default the content root is the same as application base path for the executable hosting the app; an alternative location can be specified with *WebHostBuilder*.

1.1.9 Web root

The web root of your app is the directory in your project for public, static resources like css, js, and image files. The static files middleware will only serve files from the web root directory (and sub-directories) by default. The web root path defaults to *<content root>/wwwroot*, but you can specify a different location using the *WebHostBuilder*.

1.1.10 Configuration

ASP.NET Core uses a new configuration model for handling simple name-value pairs. The new configuration model is not based on `System.Configuration` or *web.config*; rather, it pulls from an ordered set of configuration providers. The built-in configuration providers support a variety of file formats (XML, JSON, INI) and environment variables to enable environment-based configuration. You can also write your own custom configuration providers.

See *Configuration* for more information.

1.1.11 Environments

Environments, like "Development" and "Production", are a first-class notion in ASP.NET Core and can be set using environment variables. See *Working with Multiple Environments* for more information.

1.1.12 Build web UI and web APIs using ASP.NET Core MVC

- You can create well-factored and testable web apps that follow the Model-View-Controller (MVC) pattern. See *MVC* and *Testing*.

- You can build HTTP services that support multiple formats and have full support for content negotiation. See *Formatting Response Data*

- *Razor* provides a productive language to create *Views*

- *Tag Helpers* enable server-side code to participate in creating and rendering HTML elements in Razor files

- You can create HTTP services with full support for content negotiation using custom or built-in formatters (JSON, XML)

- *Model Binding* automatically maps data from HTTP requests to action method parameters

- *Model Validation* automatically performs client and server side validation

1.1.13 Client-side development

ASP.NET Core is designed to integrate seamlessly with a variety of client-side frameworks, including *AngularJS*, *KnockoutJS* and *Bootstrap*. See *Client-Side Development* for more details.

1.1.14 Next steps

- *Building your first ASP.NET Core MVC app with Visual Studio*

- *Your First ASP.NET Core Application on a Mac Using Visual Studio Code*

- *Building Your First Web API with ASP.NET Core MVC and Visual Studio*

- *Fundamentals*

1.2 Getting Started

1. Install .NET Core

2. Create a new .NET Core project:

```
mkdir aspnetcoreapp
cd aspnetcoreapp
dotnet new
```

3. Update the *project.json* file to add the Kestrel HTTP server package as a dependency:

```
{
  "version": "1.0.0-*",
  "buildOptions": {
    "emitEntryPoint": true
  },
  "dependencies": {
    "Microsoft.NETCore.App": {
      "type": "platform",
      "version": "1.0.0"
    },
    "Microsoft.AspNetCore.Server.Kestrel": "1.0.0"
```

```
    },
    "frameworks": {
      "netcoreapp1.0": { }
    }
  }
```

4. Restore the packages:

```
dotnet restore
```

5. Add a *Startup.cs* file that defines the request handling logic:

```csharp
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;

namespace aspnetcoreapp
{
    public class Startup
    {
        public void Configure(IApplicationBuilder app)
        {
            app.Run(context =>
            {
                return context.Response.WriteAsync("Hello from ASP.NET Core!");
            });
        }
    }
}
```

6. Update the code in *Program.cs* to setup and start the Web host:

```csharp
using System;
using Microsoft.AspNetCore.Hosting;

namespace aspnetcoreapp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}
```

7. Run the app (the `dotnet run` command will build the app when it's out of date):

```
dotnet run
```

8. Browse to http://localhost:5000:

Hello from ASP.NET Core!
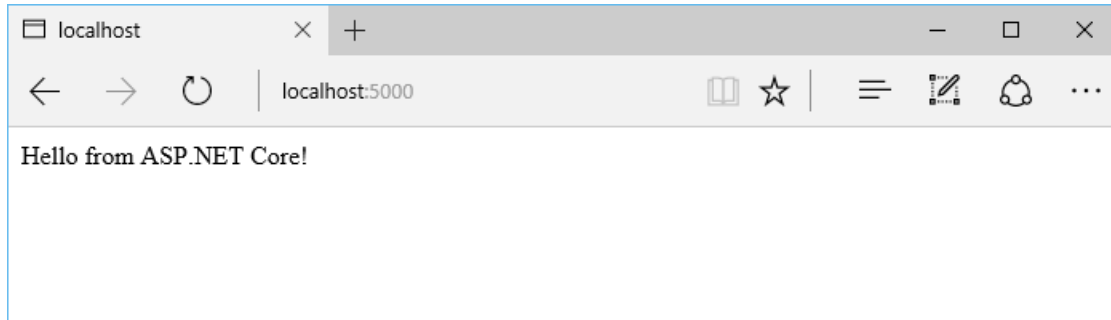
1.2.1 Next steps

- *Building your first ASP.NET Core MVC app with Visual Studio*
- *Your First ASP.NET Core Application on a Mac Using Visual Studio Code*
- *Building Your First Web API with ASP.NET Core MVC and Visual Studio*
- *Fundamentals*

1.3 Tutorials

1.3.1 Your First ASP.NET Core Application on a Mac Using Visual Studio Code

By Daniel Roth, Steve Smith and Rick Anderson

This article will show you how to write your first ASP.NET Core application on a Mac.

Sections:

- *Setting Up Your Development Environment*
- *Scaffolding Applications Using Yeoman*
- *Developing ASP.NET Core Applications on a Mac With Visual Studio Code*
- *Running Locally Using Kestrel*
- *Publishing to Azure*
- *Additional Resources*

**Setting Up Your Development Environment**

To setup your development machine download and install .NET Core and Visual Studio Code with the C# extension.
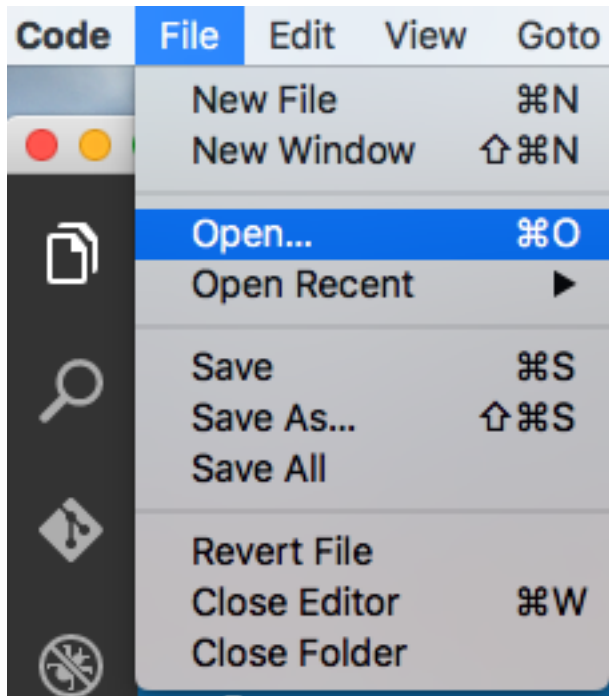
Scaffolding Applications Using Yeoman

Follow the instruction in *Building Projects with Yeoman* to create an ASP.NET Core project.

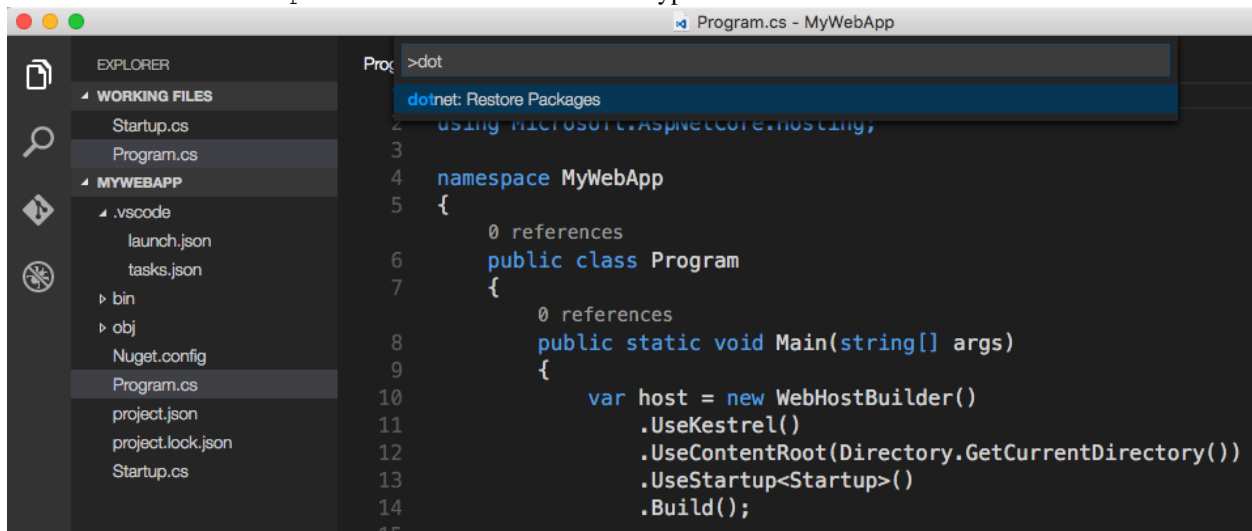## Developing ASP.NET Core Applications on a Mac With Visual Studio Code

- Start **Visual Studio Code**
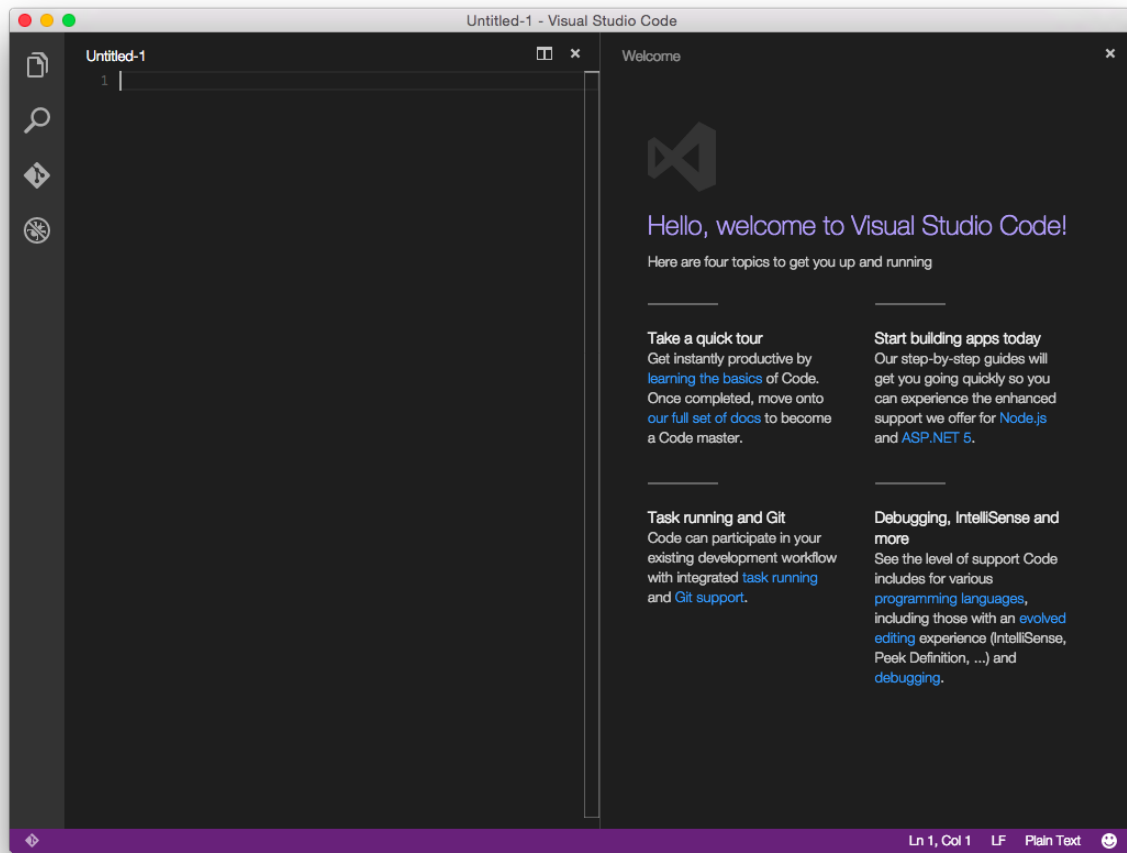
- Tap **File > Open** and navigate to your Empty ASP.NET Core app

From a Terminal / bash prompt, run `dotnet restore` to restore the project's dependencies. Alternately, you can enter `command shift p` in Visual Studio Code and then type `dot` as shown:

You can run commands directly from within Visual Studio Code, including `dotnet restore` and any tools referenced in the *project.json* file, as well as custom tasks defined in *.vscode/tasks.json*.

This empty project template simply displays "Hello World!". Open *Startup.cs* in Visual Studio Code to see how this is configured:
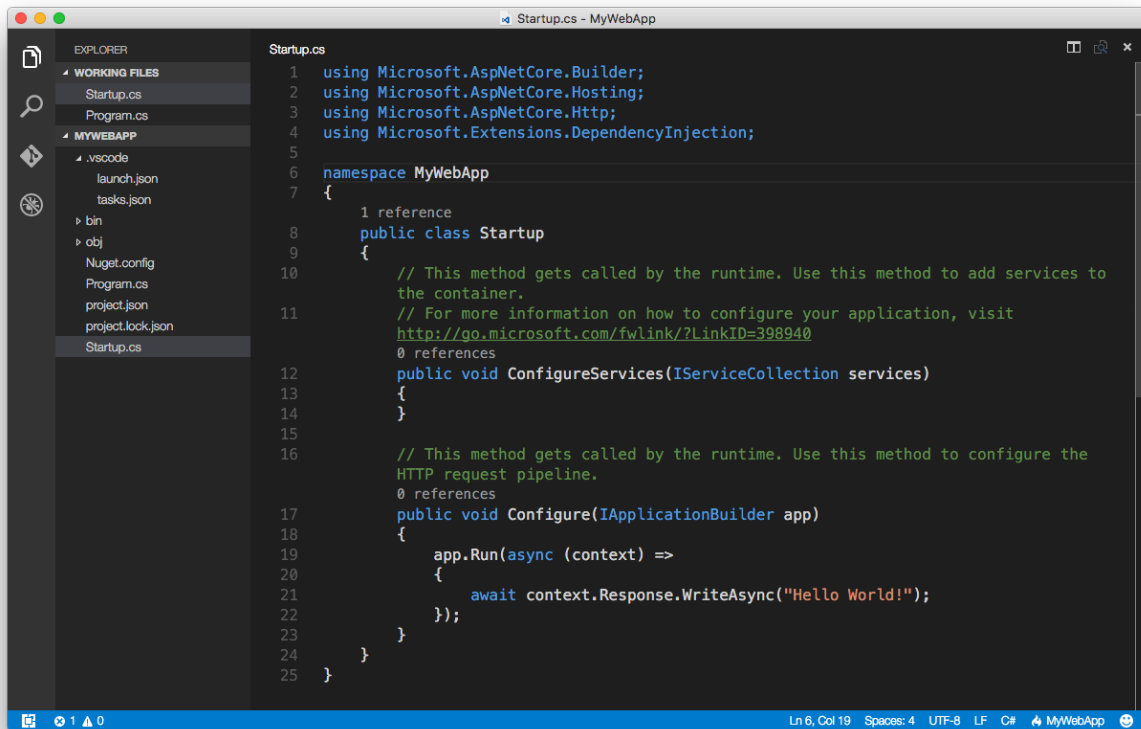
If this is your first time using Visual Studio Code (or just *Code* for short), note that it provides a very streamlined, fast, clean interface for quickly working with files, while still providing tooling to make writing code extremely productive.

In the left navigation bar, there are four icons, representing four viewlets:

- Explore

- Search

- Git

- Debug

The Explore viewlet allows you to quickly navigate within the folder system, as well as easily see the files you are currently working with. It displays a badge to indicate whether any files have unsaved changes, and new folders and files can easily be created (without having to open a separate dialog window). You can easily Save All from a menu option that appears on mouse over, as well.

The Search viewlet allows you to quickly search within the folder structure, searching filenames as well as contents.

Code will integrate with Git if it is installed on your system. You can easily initialize a new repository, make commits, and push changes from the Git viewlet.
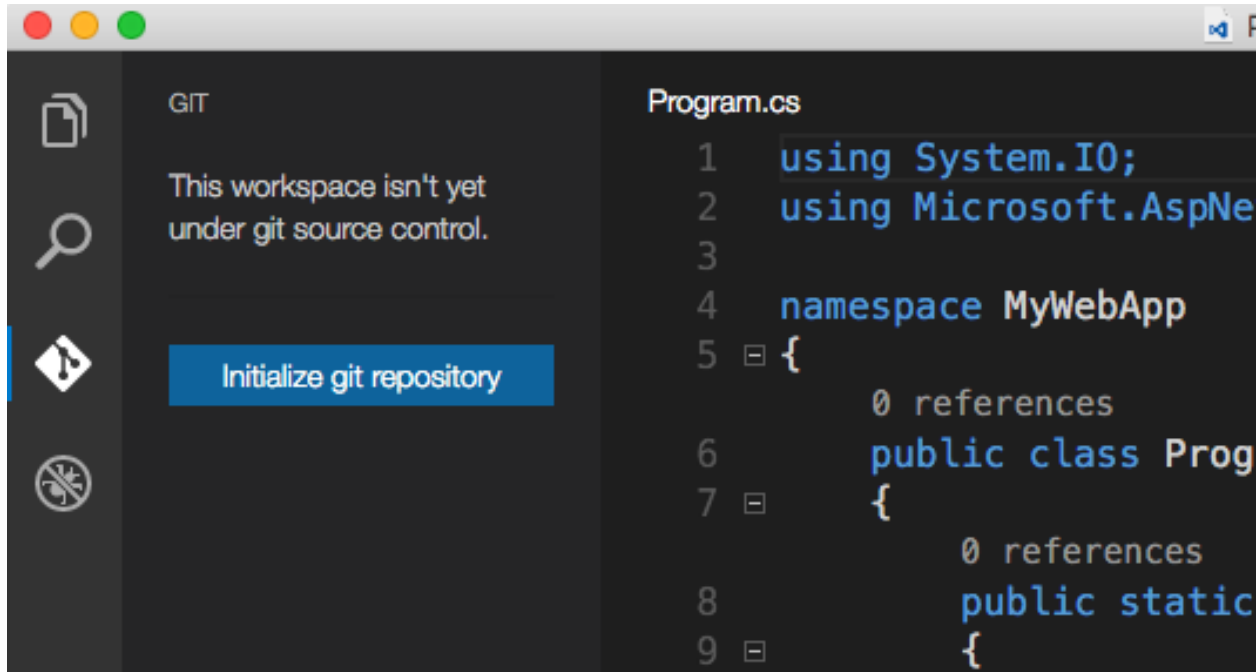
The Debug viewlet supports interactive debugging of applications.

Finally, Code's editor has a ton of great features. You'll notice unused using statements are underlined and can be removed automatically by using `command .` when the lightbulb icon appears. Classes and methods also display how many references there are in the project to them. If you're coming from Visual Studio, Code includes many of the same keyboard shortcuts, such as `command k c` to comment a block of code, and `command k u` to uncomment.

Running Locally Using Kestrel

The sample is configured to use *Kestrel* for the web server. You can see it configured in the *project.json* file, where it is specified as a dependency.

```json
{
  "buildOptions": {
    "emitEntryPoint": true
  },
  "dependencies": {
    "Microsoft.NETCore.App": {
      "type": "platform",
      "version": "1.0.0"
    },
    "Microsoft.AspNetCore.Server.Kestrel": "1.0.0"
  },
  "frameworks": {
    "netcoreapp1.0": {}
  }
}
```

- Run `dotnet run` command to launch the app

- Navigate to `localhost:5000`:
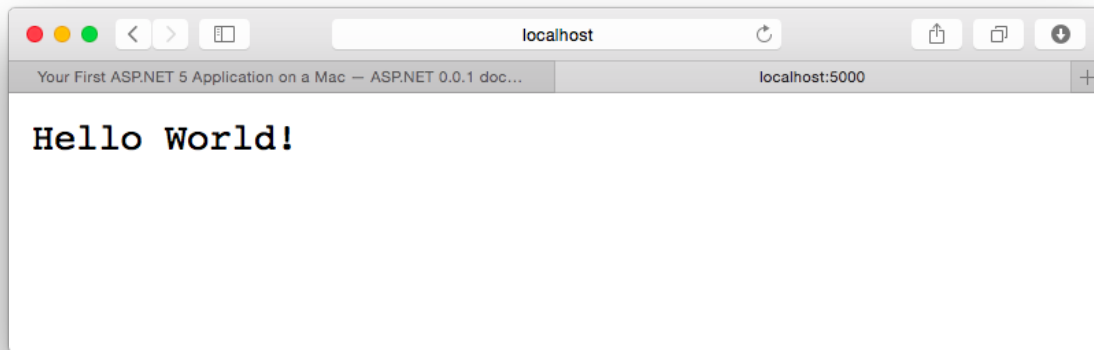
- To stop the web server enter `Ctrl+C`.

Publishing to Azure

Once you've developed your application, you can easily use the Git integration built into Visual Studio Code to push updates to production, hosted on Microsoft Azure.

Initialize Git

Initialize Git in the folder you're working in. Tap on the Git viewlet and click the `Initialize Git repository` button.
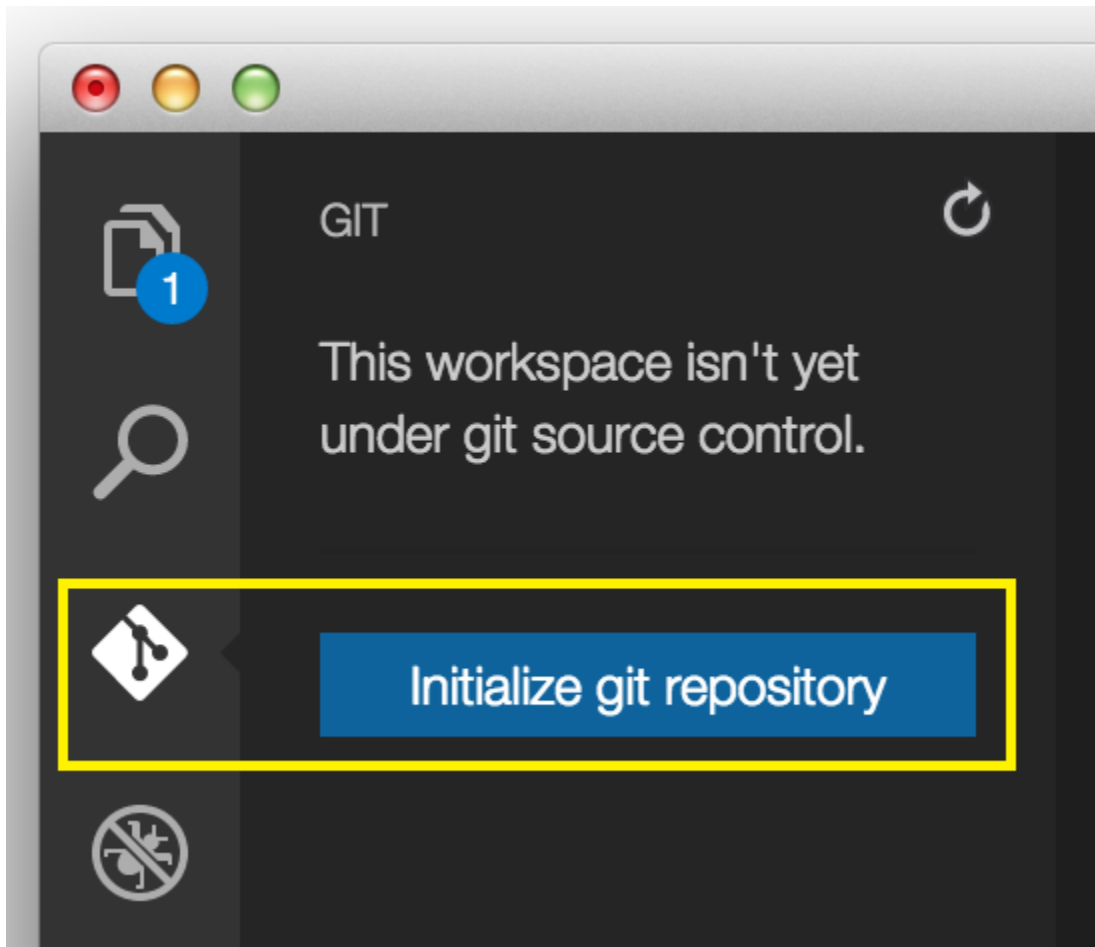
Add a commit message and tap enter or tap the checkmark icon to commit the staged files.
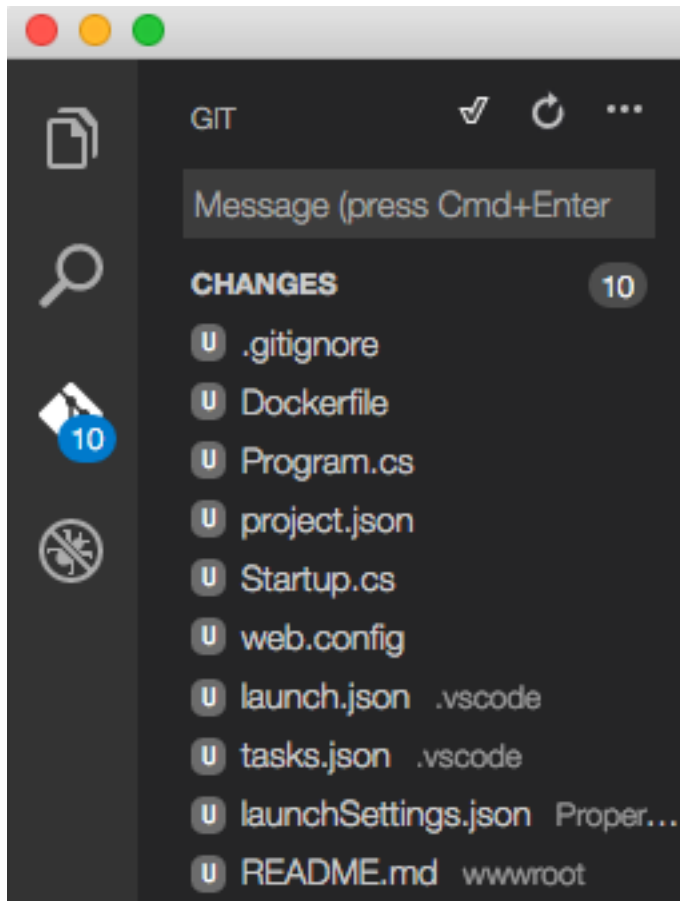
Git is tracking changes, so if you make an update to a file, the Git viewlet will display the files that have changed since your last commit.

Initialize Azure Website
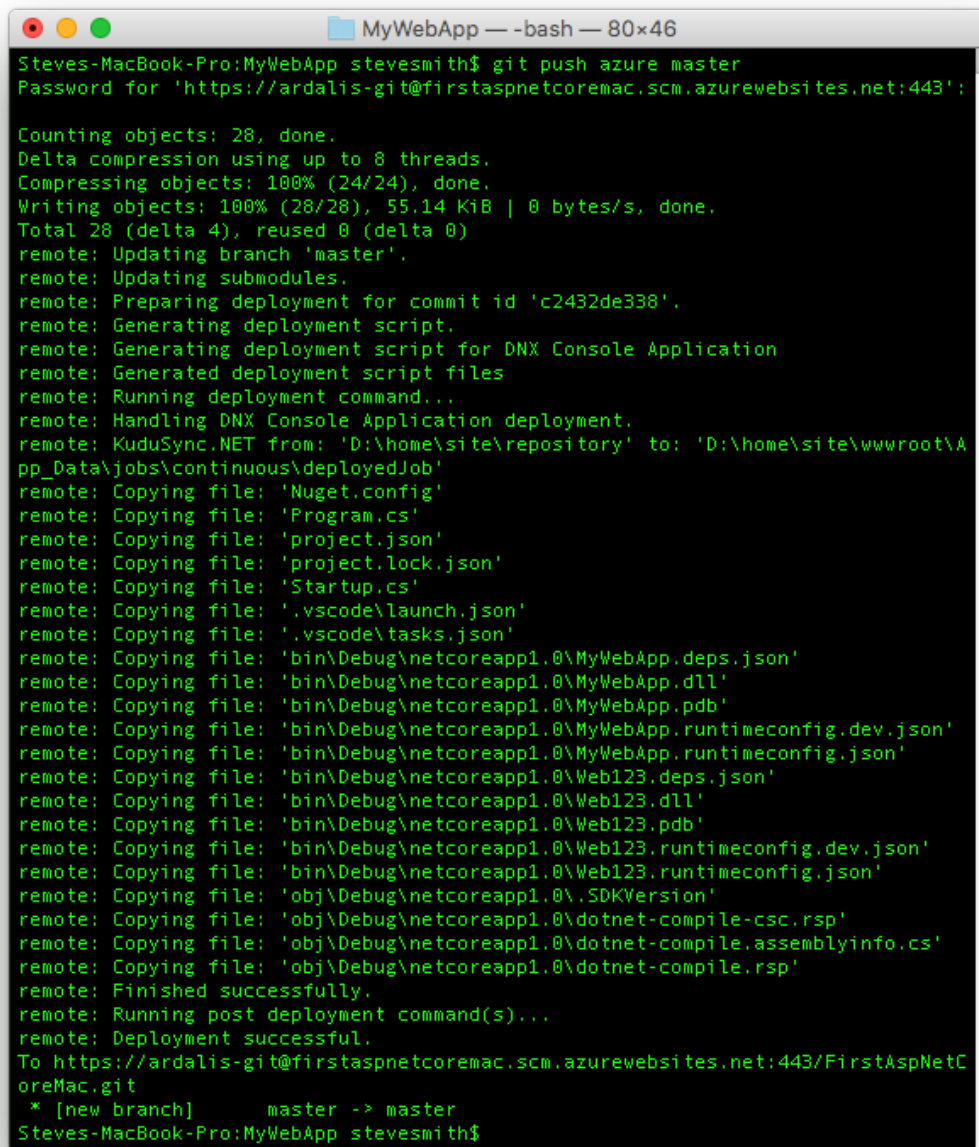
You can deploy to Azure Web Apps directly using Git.

- Create a new Web App in Azure. If you don't have an Azure account, you can create a free trial.
- Configure the Web App in Azure to support continuous deployment using Git.

Record the Git URL for the Web App from the Azure portal:
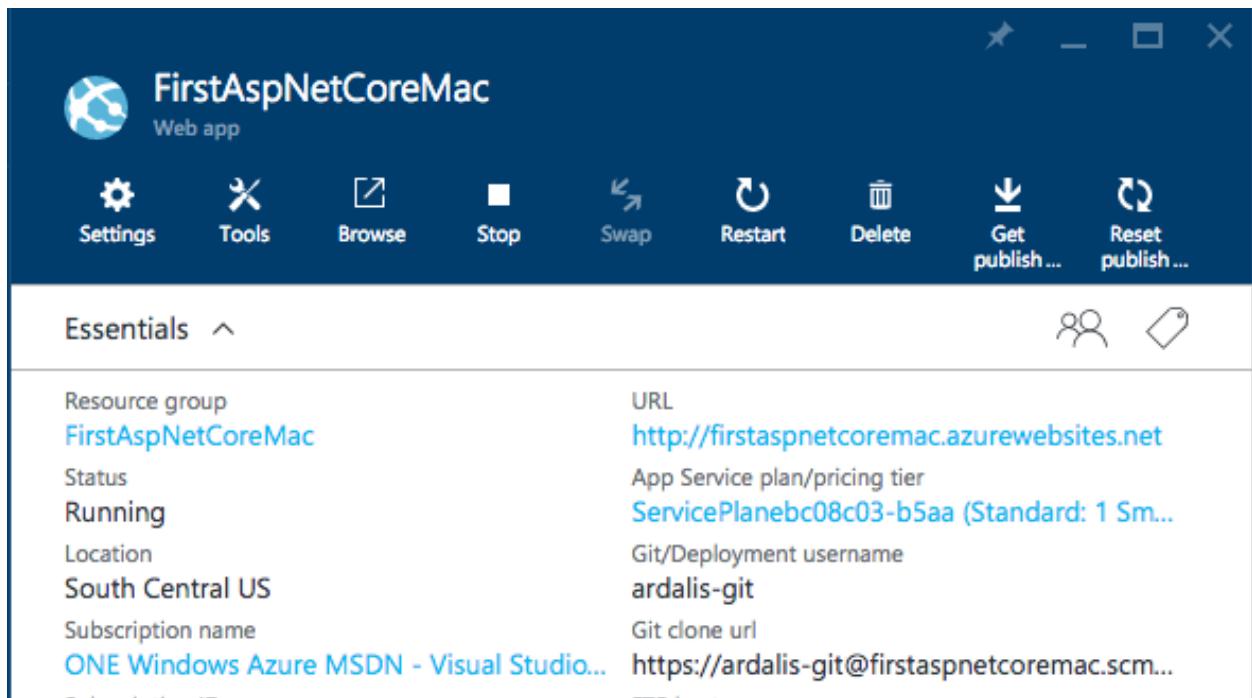
- In a Terminal window, add a remote named `azure` with the Git URL you noted previously.

    - `git remote add azure https://ardalis-git@firstaspnetcoremac.scm.azurewebsites.net:4`

- Push to master.

    - `git push azure master` to deploy.

```
Steves-MacBook-Pro:MyWebApp stevesmith$ git push azure master
Password for 'https://ardalis-git@firstaspnetcoremac.scm.azurewebsites.net:443':

Counting objects: 28, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (24/24), done.
Writing objects: 100% (28/28), 55.14 KiB | 0 bytes/s, done.
Total 28 (delta 4), reused 0 (delta 0)
remote: Updating branch 'master'.
remote: Updating submodules.
remote: Preparing deployment for commit id 'c2432de338'.
remote: Generating deployment script.
remote: Generating deployment script for DNX Console Application
remote: Generated deployment script files
remote: Running deployment command...
remote: Handling DNX Console Application deployment.
remote: KuduSync.NET from: 'D:\home\site\repository' to: 'D:\home\site\wwwroot\A
pp_Data\jobs\continuous\deployedJob'
remote: Copying file: 'Nuget.config'
remote: Copying file: 'Program.cs'
remote: Copying file: 'project.json'
remote: Copying file: 'project.lock.json'
remote: Copying file: 'Startup.cs'
remote: Copying file: '.vscode\launch.json'
remote: Copying file: '.vscode\tasks.json'
remote: Copying file: 'bin\Debug\netcoreapp1.0\MyWebApp.deps.json'
remote: Copying file: 'bin\Debug\netcoreapp1.0\MyWebApp.dll'
remote: Copying file: 'bin\Debug\netcoreapp1.0\MyWebApp.pdb'
remote: Copying file: 'bin\Debug\netcoreapp1.0\MyWebApp.runtimeconfig.dev.json'
remote: Copying file: 'bin\Debug\netcoreapp1.0\MyWebApp.runtimeconfig.json'
remote: Copying file: 'bin\Debug\netcoreapp1.0\Web123.deps.json'
remote: Copying file: 'bin\Debug\netcoreapp1.0\Web123.dll'
remote: Copying file: 'bin\Debug\netcoreapp1.0\Web123.pdb'
remote: Copying file: 'bin\Debug\netcoreapp1.0\Web123.runtimeconfig.dev.json'
remote: Copying file: 'bin\Debug\netcoreapp1.0\Web123.runtimeconfig.json'
remote: Copying file: 'obj\Debug\netcoreapp1.0\.SDKVersion'
remote: Copying file: 'obj\Debug\netcoreapp1.0\dotnet-compile-csc.rsp'
remote: Copying file: 'obj\Debug\netcoreapp1.0\dotnet-compile.assemblyinfo.cs'
remote: Copying file: 'obj\Debug\netcoreapp1.0\dotnet-compile.rsp'
remote: Finished successfully.
remote: Running post deployment command(s)...
remote: Deployment successful.
To https://ardalis-git@firstaspnetcoremac.scm.azurewebsites.net:443/FirstAspNetC
oreMac.git
 * [new branch]      master -> master
Steves-MacBook-Pro:MyWebApp stevesmith$
```

- Browse to the newly deployed web app. You should see `Hello world!`

Additional Resources

- Visual Studio Code
- *Building Projects with Yeoman*
- *Fundamentals*

1.3.2 Building Your First Web API with ASP.NET Core MVC and Visual Studio

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.

ASP.NET Core has built-in support for MVC building Web APIs. 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.

: If you are porting an existing Web API app to ASP.NET Core, see *Migrating from ASP.NET Web API*

Sections:

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

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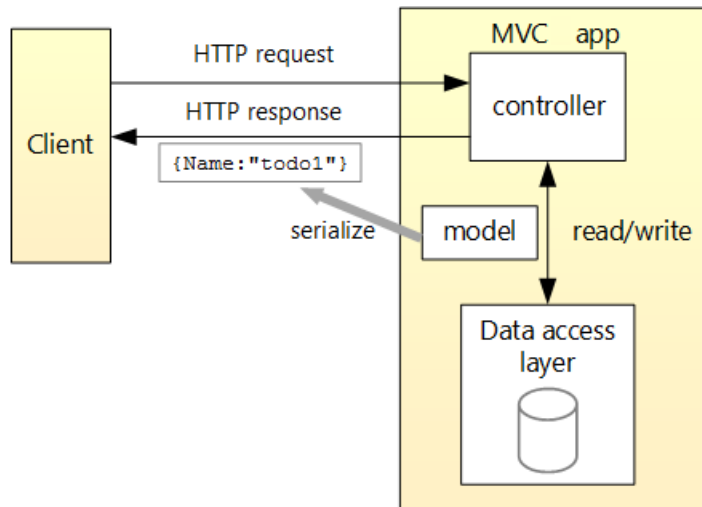
- The client is whatever consumes the web API (browser, mobile app, and so forth). We aren't writing a client in this tutorial.

- 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 the app doesn't use a database. Instead, it just keeps to-do items in memory. But we'll still include a (trivial) data access layer, to illustrate the separation between the web API and the data layer. For a tutorial that uses a database, see *Building your first ASP.NET Core MVC app with Visual Studio*.

Install Fiddler

We're not building a client, we'll use Fiddler to test the API. Fiddler is a web debugging tool that lets you compose HTTP requests and view the raw HTTP responses.

Create the project

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

Select the **ASP.NET Core Web Application** project template. Name the project `TodoApi` and tap **OK**.
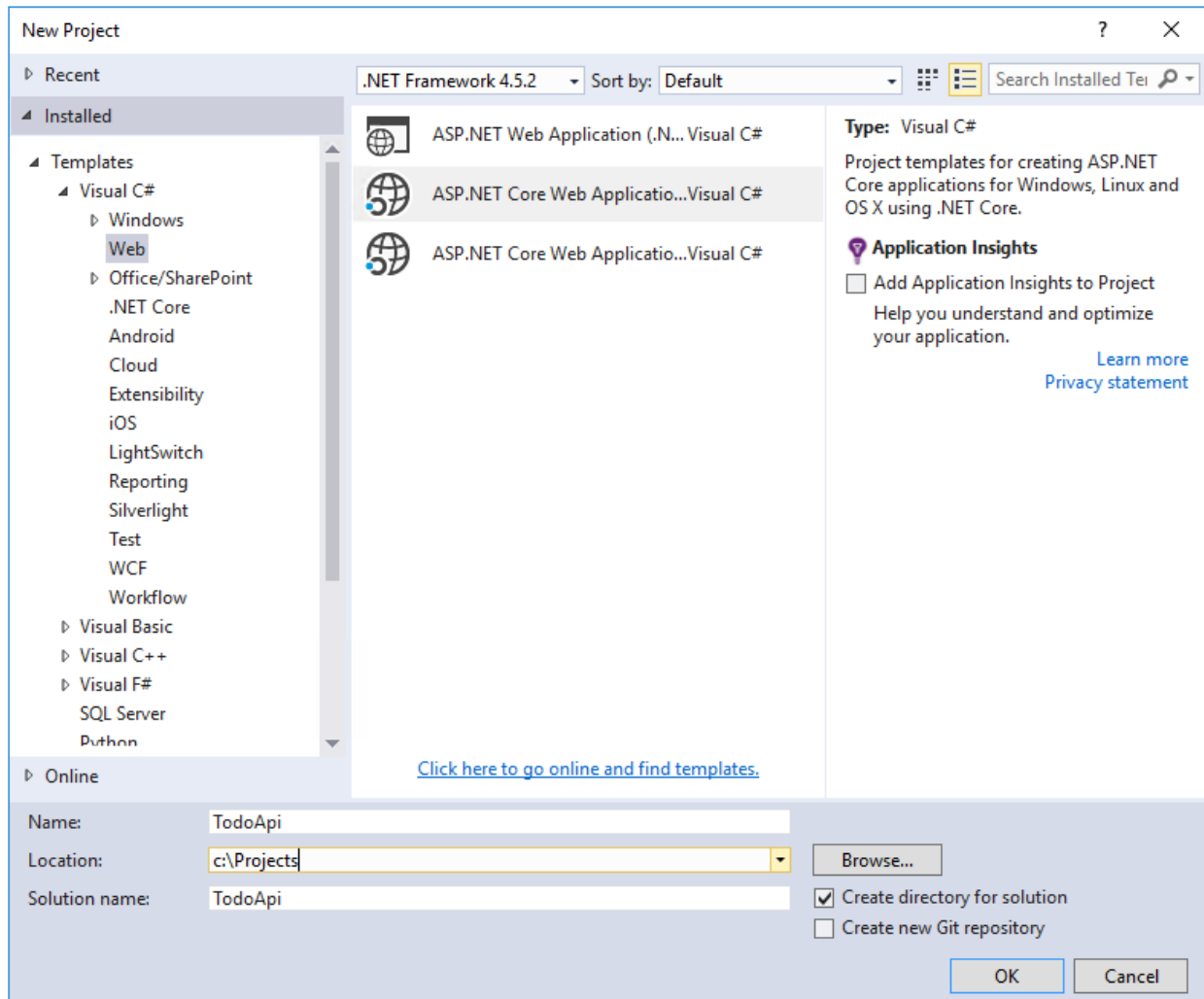
In the **New ASP.NET Core Web Application (.NET Core) - TodoApi** dialog, select the **Web API** template. Tap **OK**.
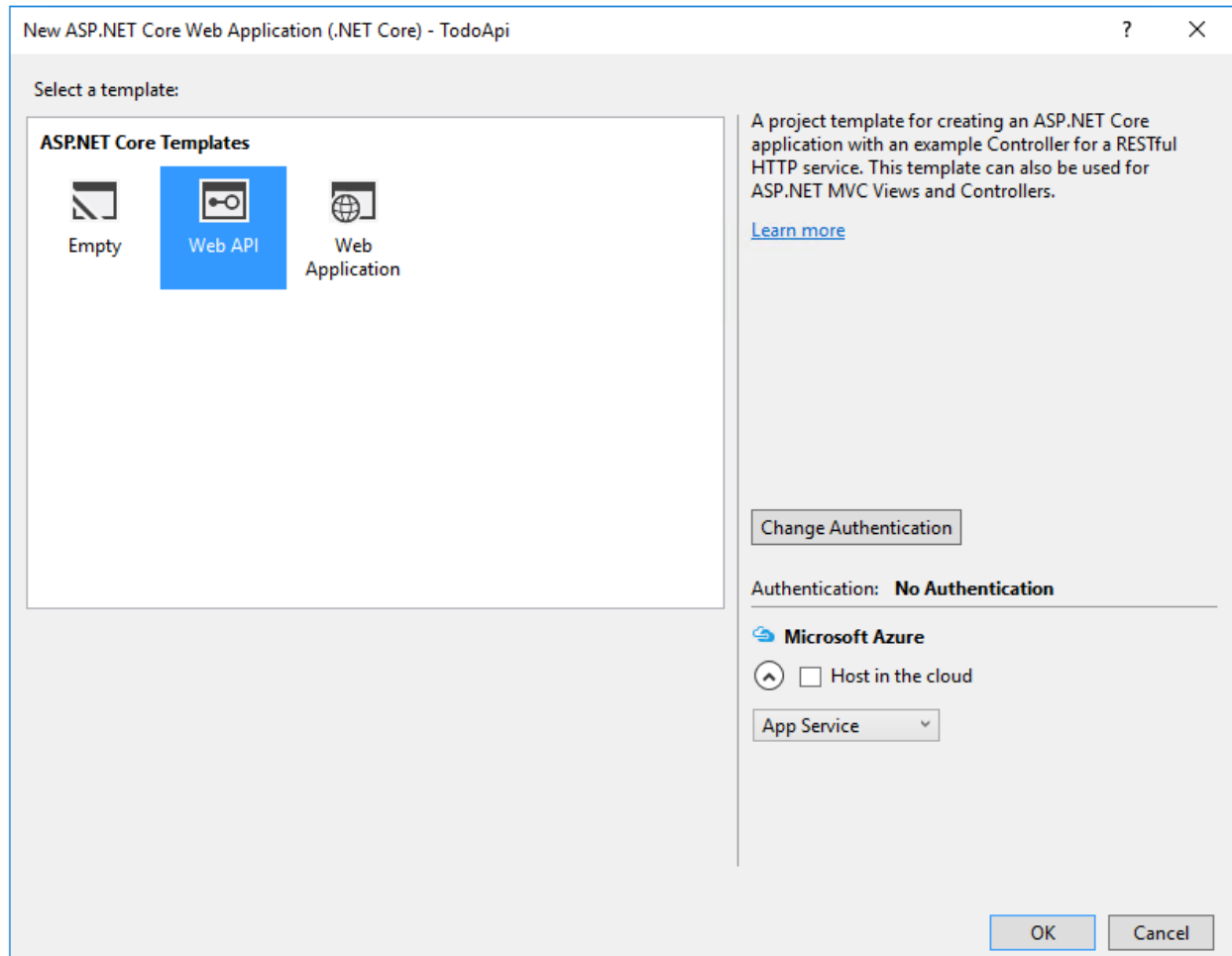
Add a model class

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

Add a folder named "Models". In Solution Explorer, right-click the project. Select **Add > New Folder**. Name the folder *Models*.
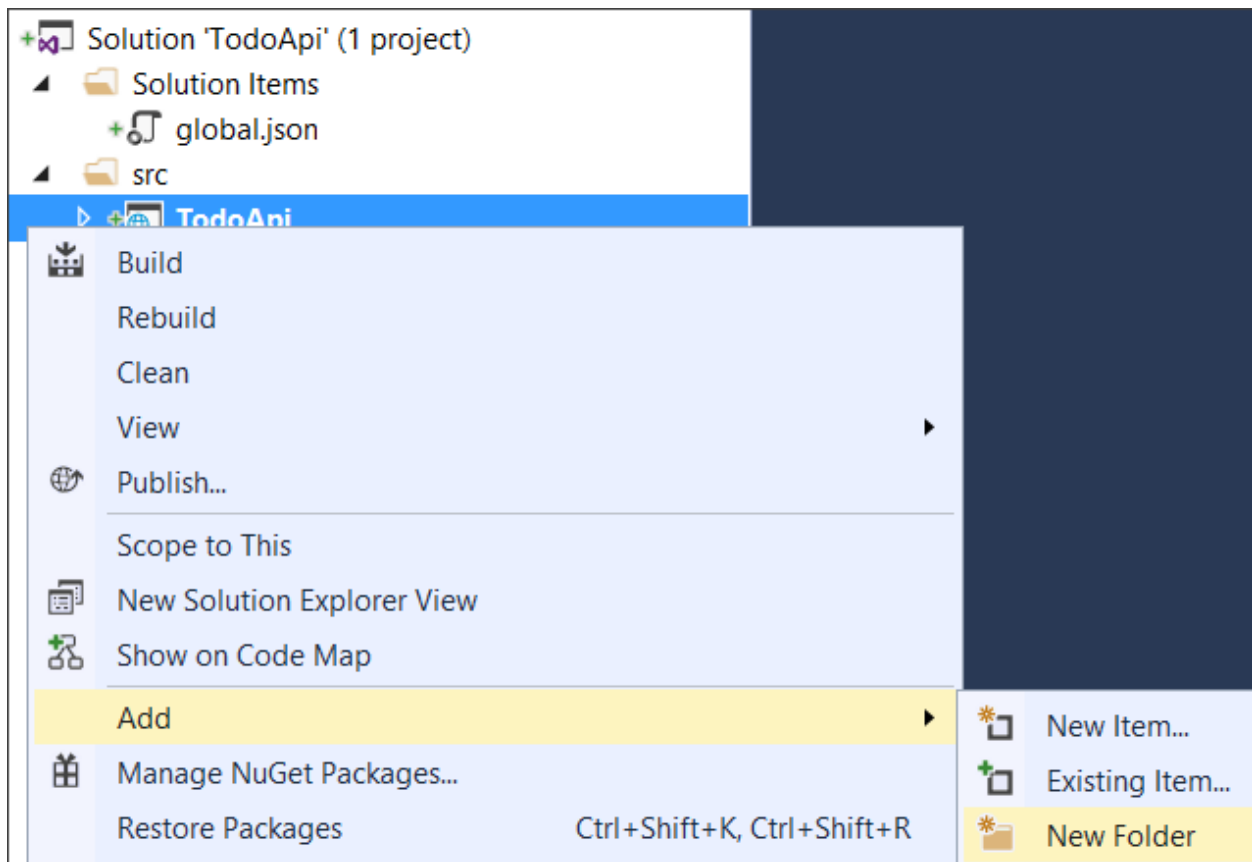
: You can put model classes anywhere in your project, but the *Models* folder is used by convention.

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

In the **Add New Item** dialog, select the **Class** template. Name the class `TodoItem` and click **OK**.
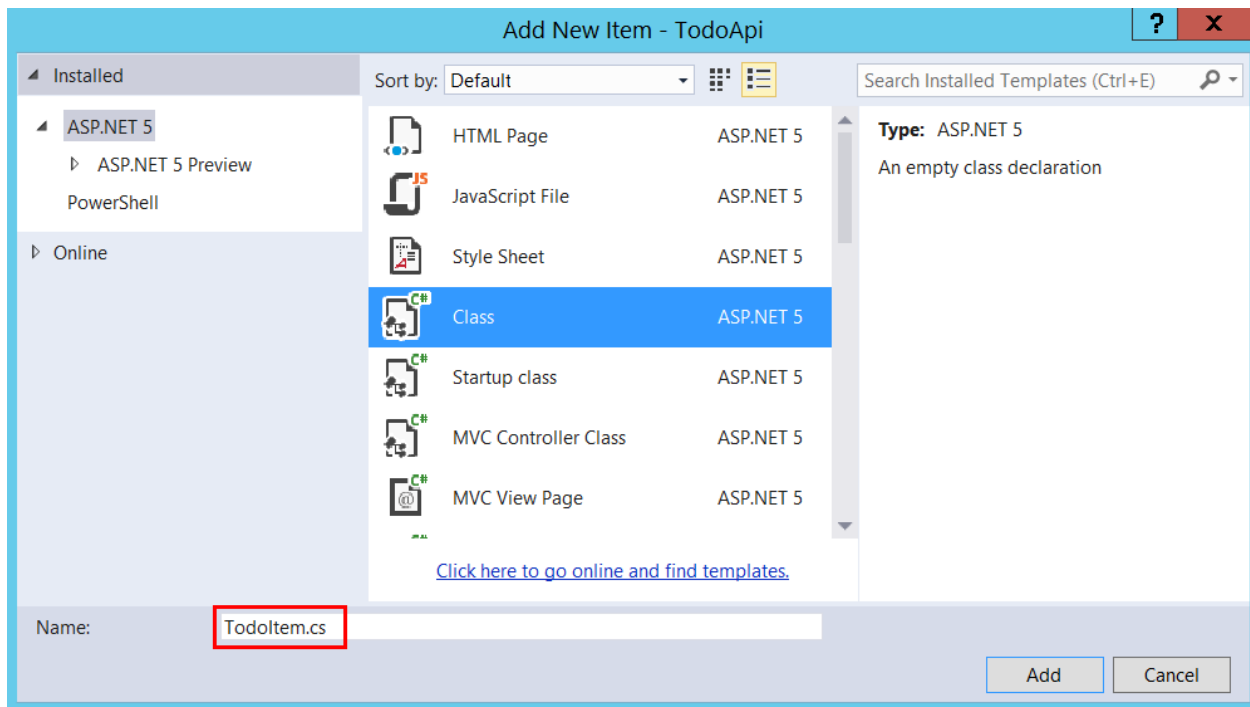
Replace the generated code with:

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

Add a repository class

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

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

```
using System.Collections.Generic;

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

This interface defines basic CRUD operations.

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

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

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

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

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

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

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

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

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

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

Register the repository

By defining a repository interface, we can decouple the repository class from the MVC controller that uses it. Instead of instantiating a `TodoRepository` inside the controller we will inject an `ITodoRepository` the built-in support in ASP.NET Core for *dependency injection*.

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

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

```
using TodoApi.Models;
```

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

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

    services.AddLogging();

    // Add our repository type
    services.AddSingleton<ITodoRepository, TodoRepository>();
}
```

Add a controller

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

Replace the generated code with the following:

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

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

This defines an empty controller class. In the next sections, we'll add methods to implement the API.

Getting to-do items

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

```
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 NotFound();
    }
    return new ObjectResult(item);
}
```

These methods implement the two GET methods:

- `GET /api/todo`

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

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

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

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

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

Routing and URL paths

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

- Take the template string in the controller's route attribute, `[Route("api/[controller]")]`

- Replace "[Controller]" with the name of the controller, which is the controller class name minus the "Controller" suffix. For this sample the name of the controller is "todo" (case insensitive). For this sample, the controller class name is **Todo**Controller and the root name is "todo". ASP.NET MVC Core is not case sensitive.

- If the [HttpGet] attribute also has a template string, append that to the path. This sample doesn't use a template string.

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

Return values

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

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

- If no item matches the requested ID, the method returns a 404 error. This is done by returning `NotFound`.

- Otherwise, the method returns 200 with a JSON response body. This is done by returning an ObjectResult.

Use Fiddler to call the API

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

Launch Fiddler. From the **File** menu, uncheck the **Capture Traffic** option. This turns off capturing HTTP traffic.
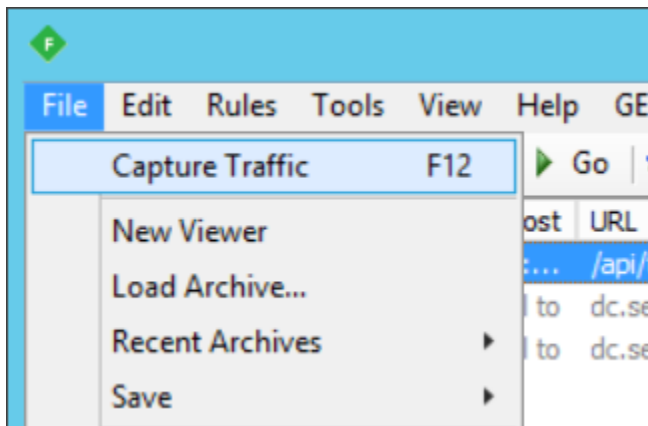
Select the **Composer** page. In the **Parsed** tab, type `http://localhost:port/api/todo`, where *port* is the port number. Click **Execute** to send the request.
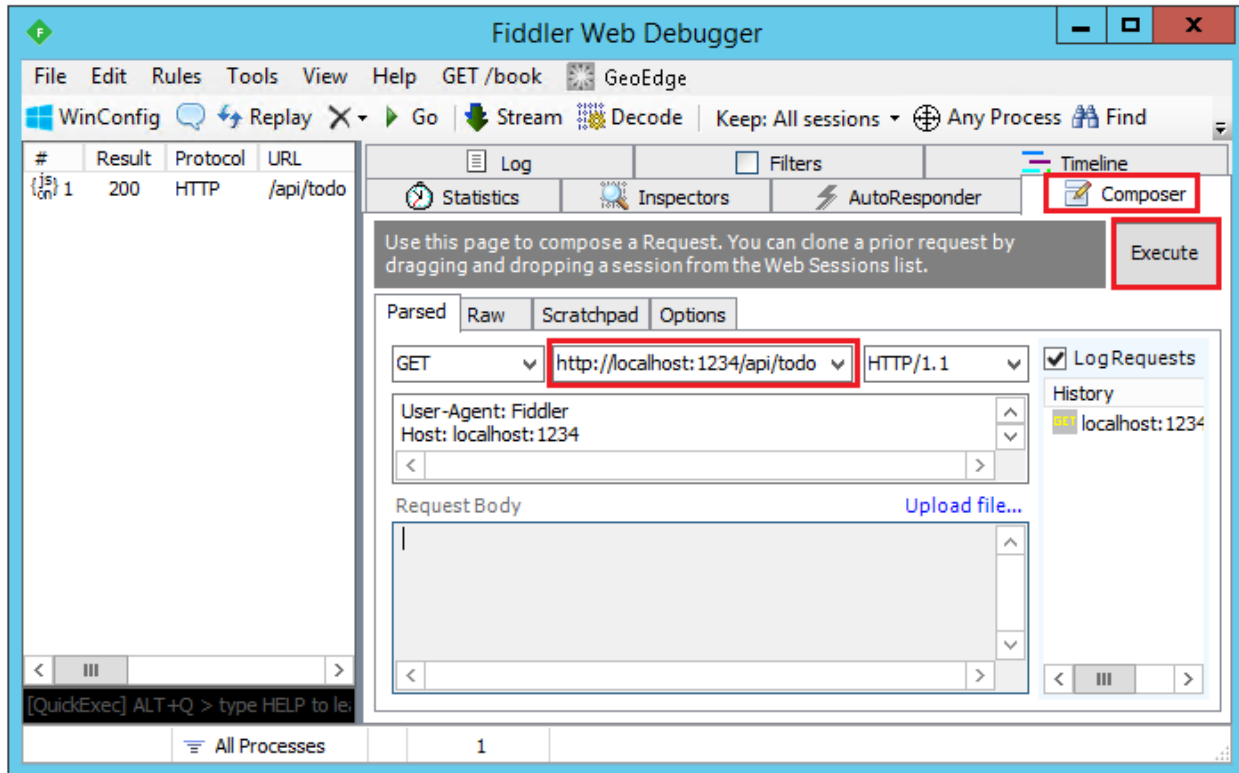
The result appears in the sessions list. The response code should be 200. Use the **Inspectors** tab to view the content of the response, including the response body.
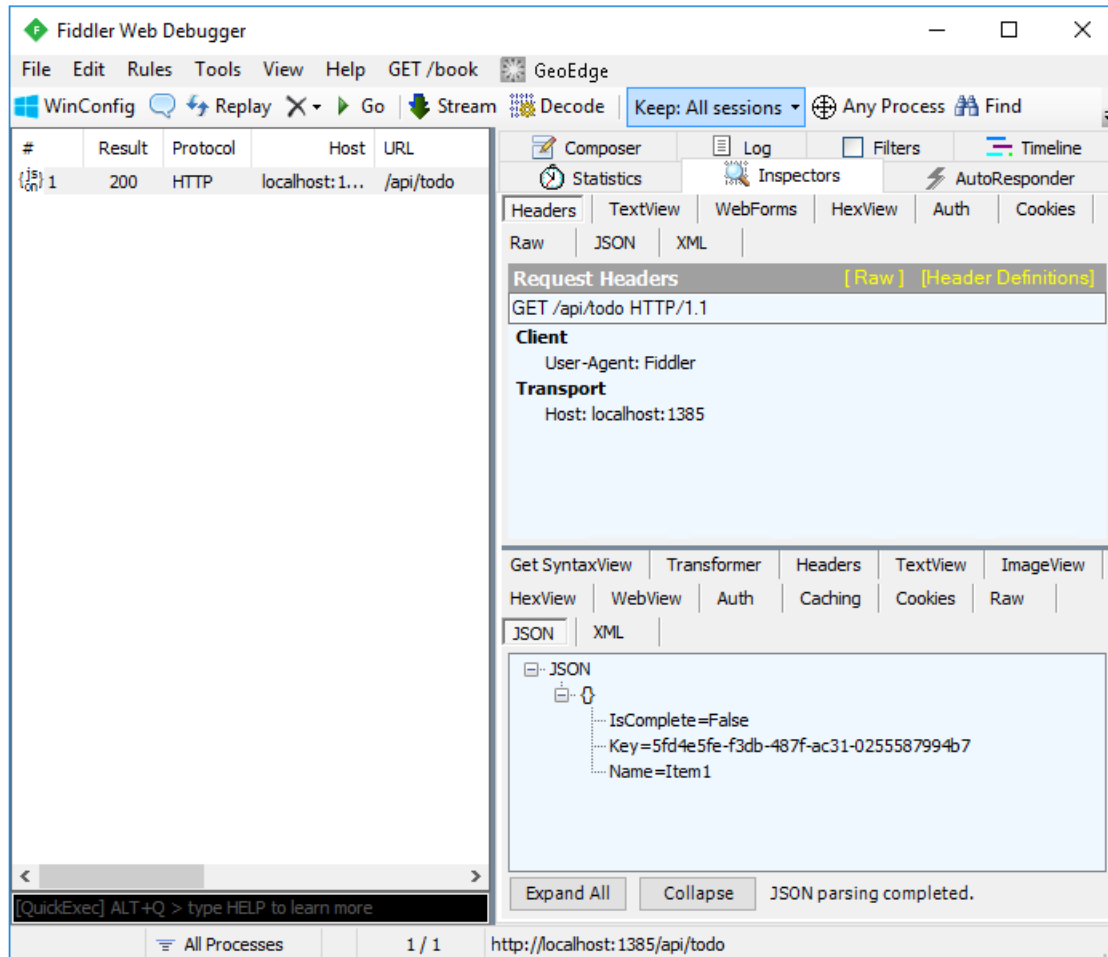
Implement the other CRUD operations

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 BadRequest();
    }
    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.

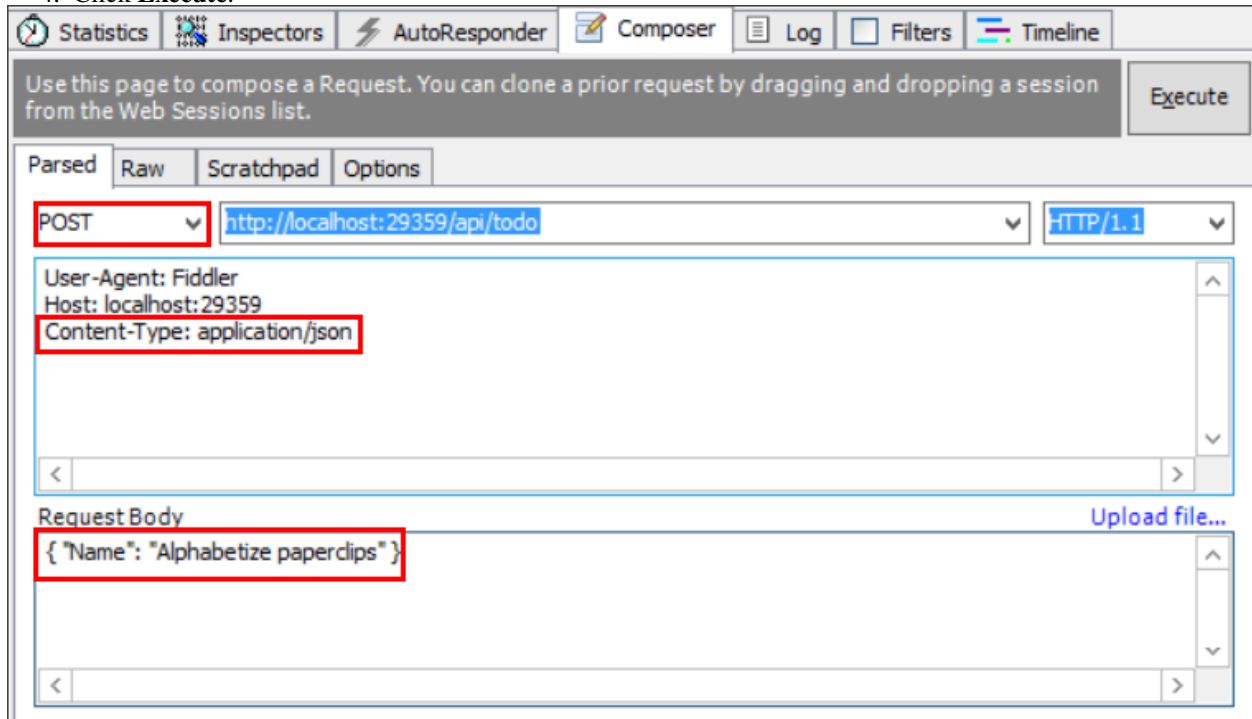2. In the request headers text box, add `Content-Type: application/json`, which is a `Content-Type` header with the value `application/json`. Fiddler automatically adds the Content-Length header.

3. In the request body text box, enter the following: `{"Name":"<your to-do item>"}`

4. Click **Execute**.

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

Request:

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

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

Response:

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

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

Update

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

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

    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.

| Parsed | Raw | Scratchpad | Options |

| PUT | ▼ | http://localhost:1234/api/todo/707e881d-ca69-40ef-b6a1-3e2fbfa ▼ | HTTP/1.1 ▼ |

```
User-Agent: Fiddler
Host: localhost:1234
Content-Type: application/json
Content-Length: 79
```

Request Body   Upload file...

```
{"Key":"707e881d-ca69-40ef-b6a1-3e2fbfa31ba6","Name":"Item1","IsComplete":true}
```
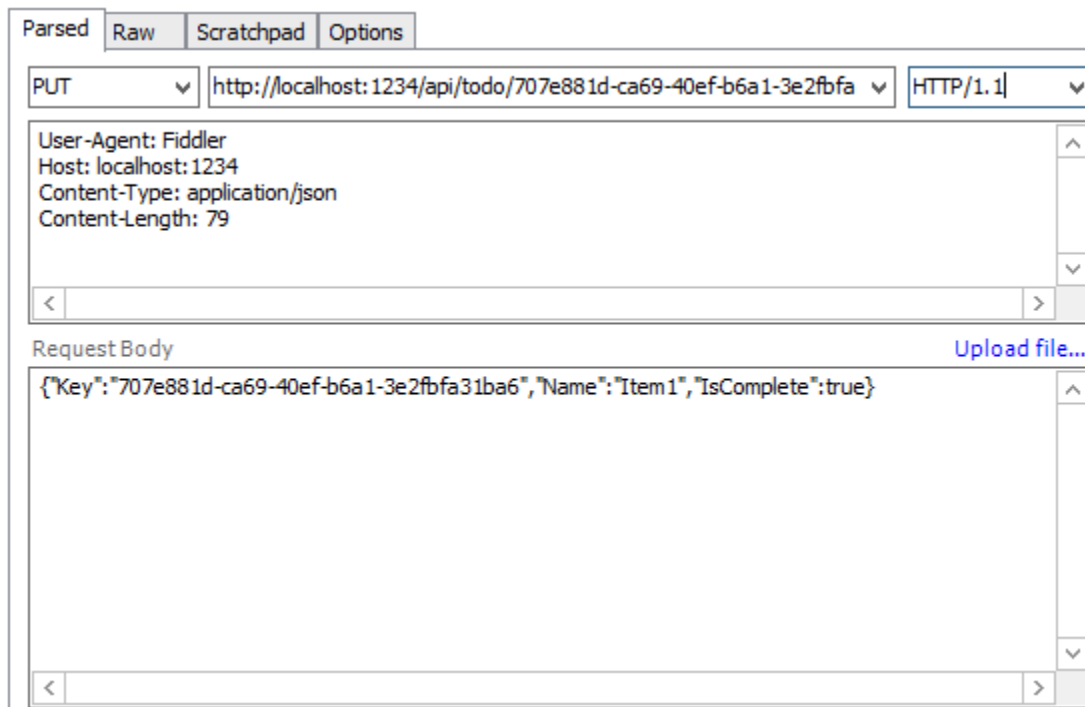
Delete

```csharp
[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.
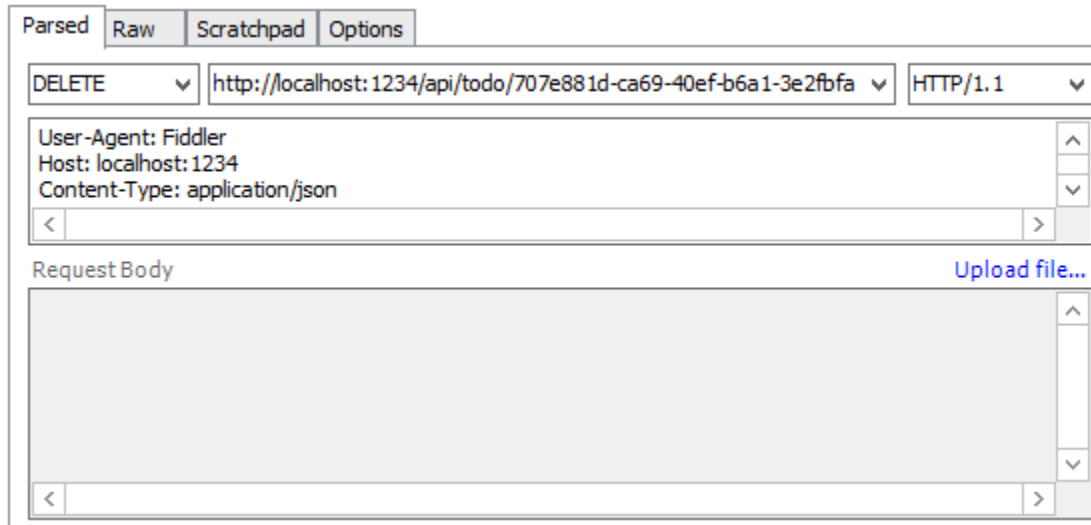
Next steps

- 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*.

- View or download sample code

1.3.3 Deploy an ASP.NET Core web app to Azure using Visual Studio

By Rick Anderson, Cesar Blum Silveira

> **Sections:**
> - *Set up the development environment*
> - *Create a web app*
> - *Test the app locally*
> - *Deploy the app to Azure*

Set up the development environment

- Install the latest Azure SDK for Visual Studio 2015. The SDK installs Visual Studio 2015 if you don't already have it.

: The SDK installation can take more than 30 minutes if your machine doesn't have many of the dependencies.

- Install .NET Core + Visual Studio tooling
- Verify your Azure account. You can open a free Azure account or Activate Visual Studio subscriber benefits.

Create a web app

In the Visual Studio Start Page, tap **New Project...**.
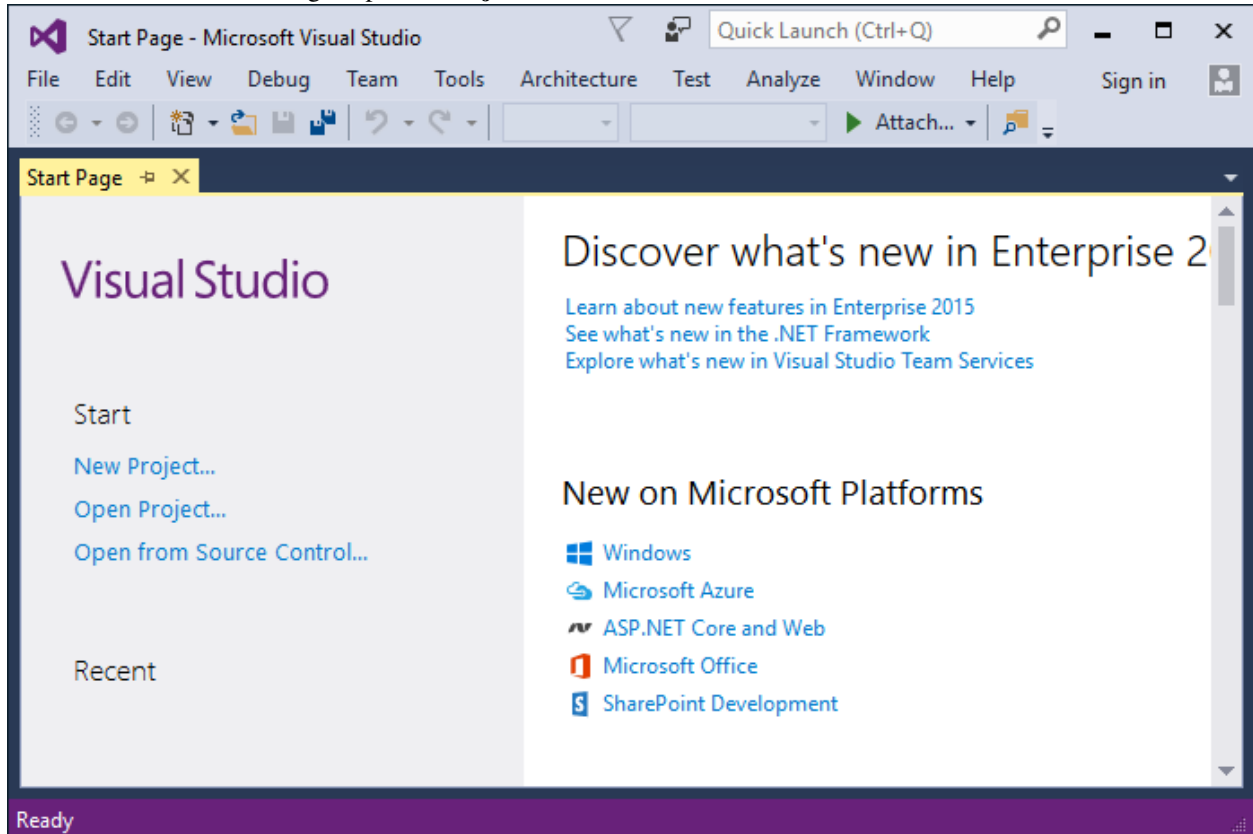
Alternatively, you can use the menus to create a new project. Tap **File > New > Project...**.
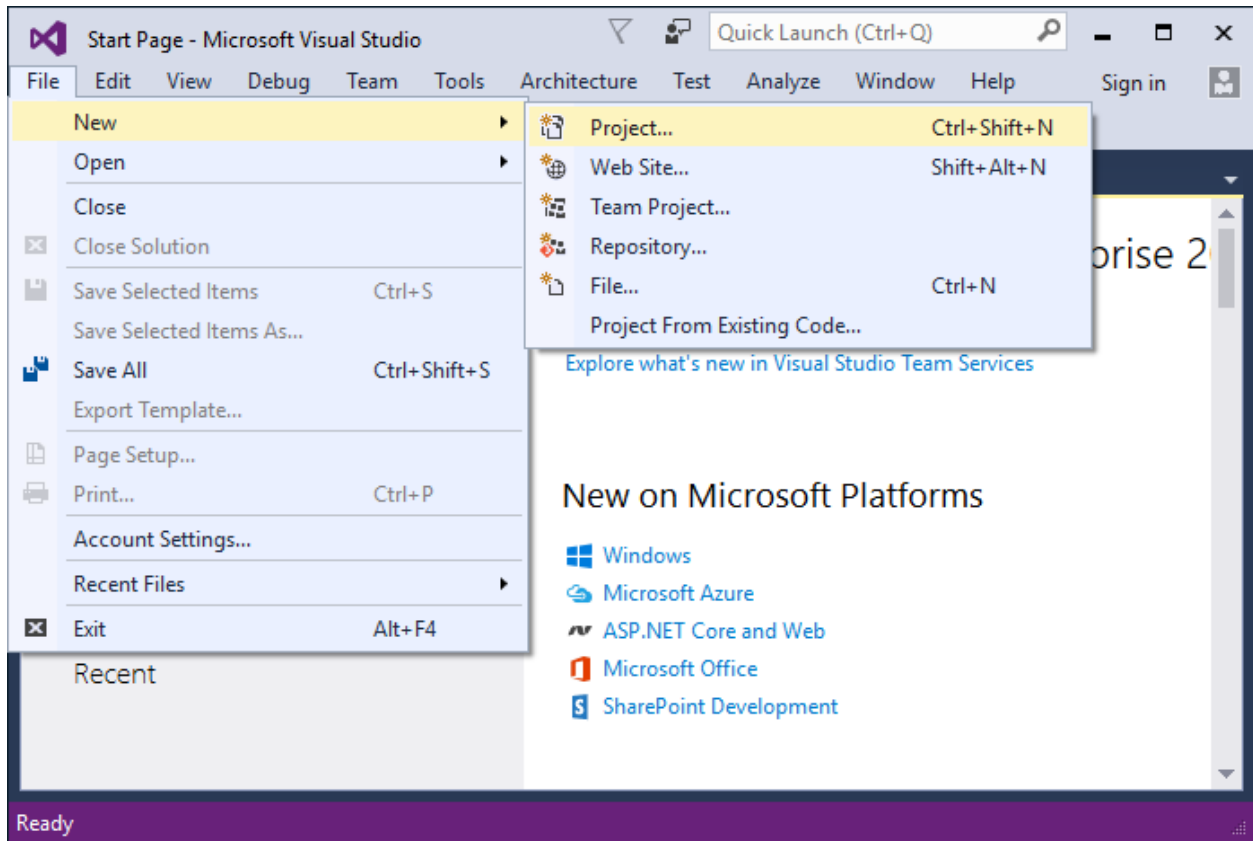
Complete the **New Project** dialog:

- In the left pane, tap **Web**

- In the center pane, tap **ASP.NET Core Web Application (.NET Core)**

- Tap **OK**
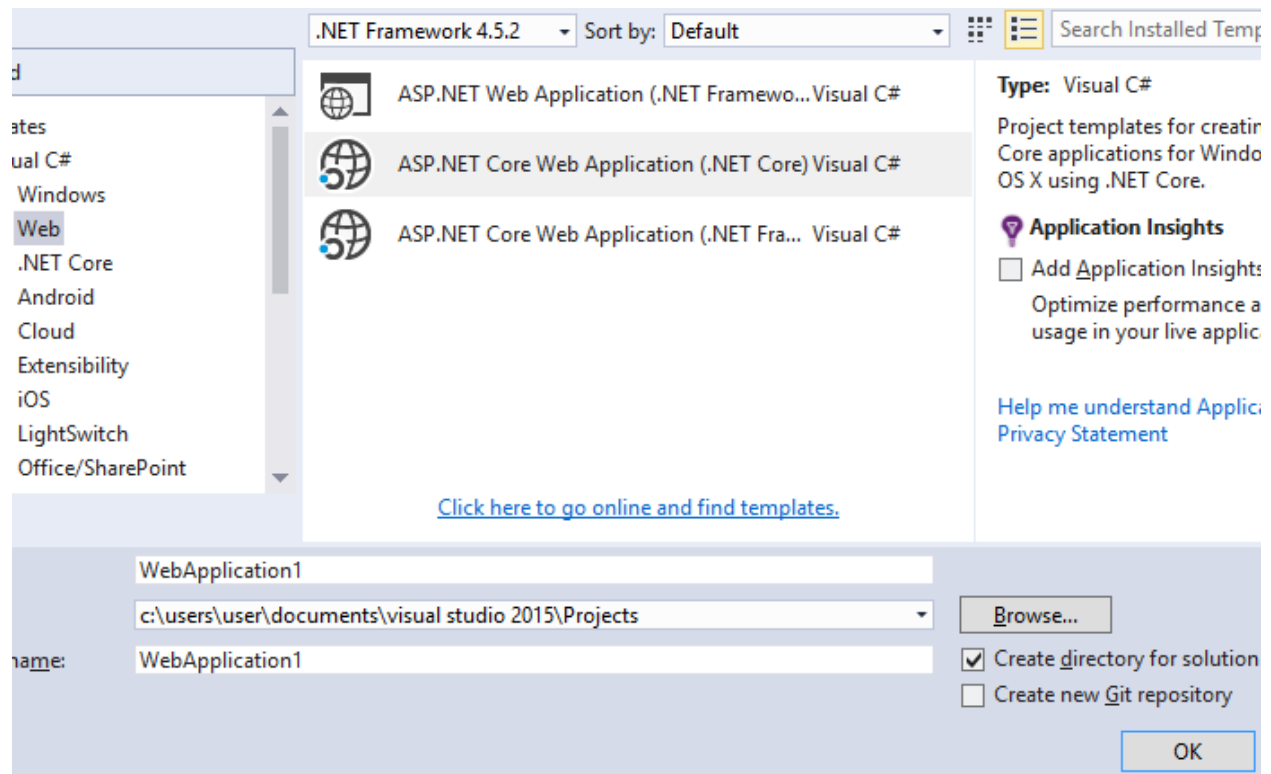
In the **New ASP.NET Core Web Application (.NET Core)** dialog:

- Tap **Web Application**
- Verify **Authentication** is set to **Individual User Accounts**
- Verify **Host in the cloud** is **not** checked
- Tap **OK**
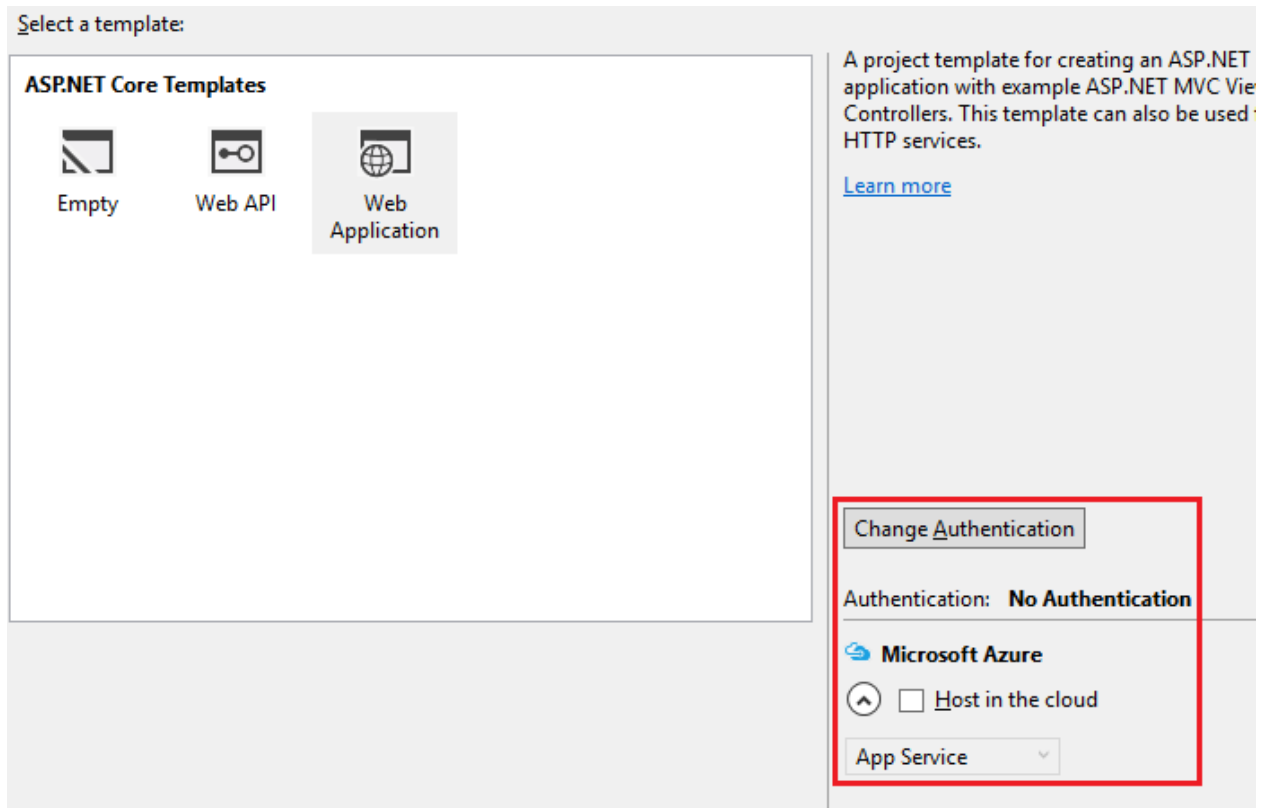
Test the app locally

- Press **Ctrl-F5** to run the app locally

- Tap the **About** and **Contact** links. Depending on the size of your device, you might need to tap the navigation icon to show the links
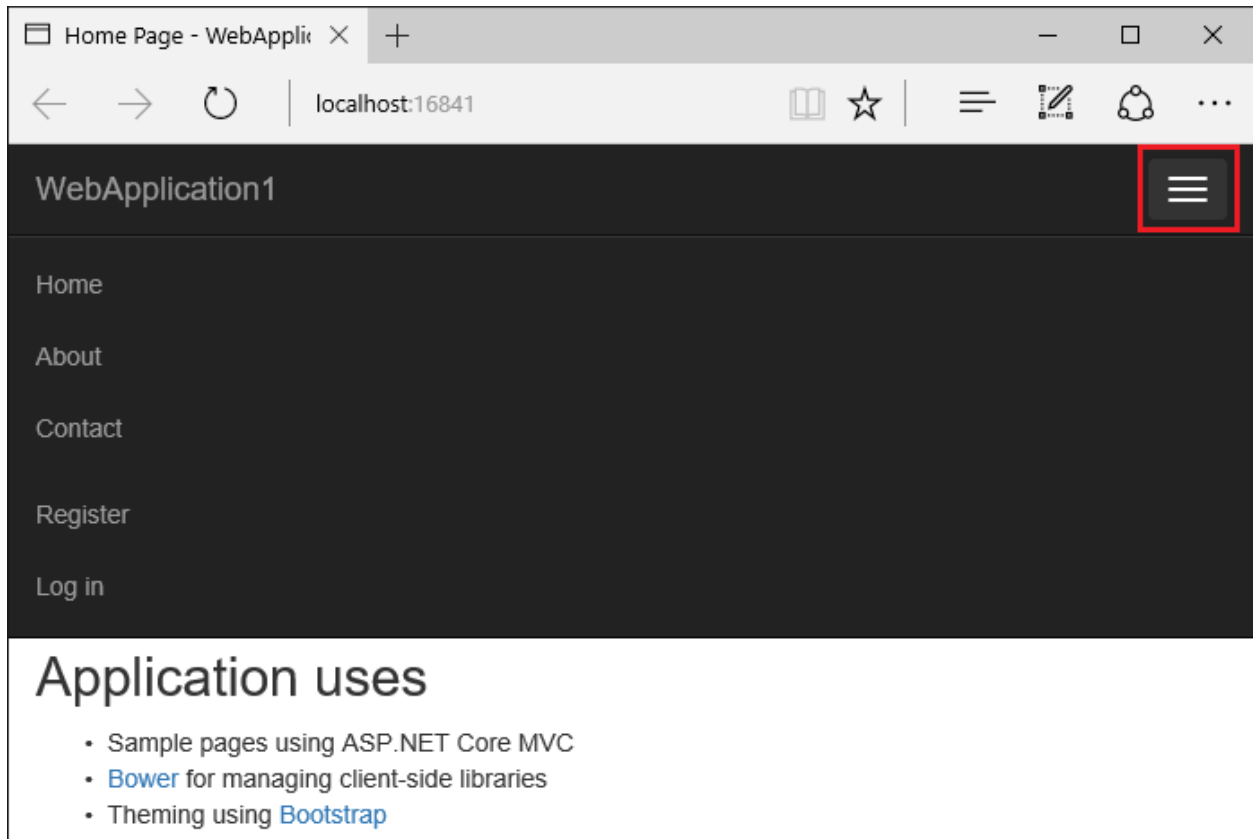
- Tap **Register** and register a new user. You can use a fictitious email address. When you submit, you'll get the following error:

You can fix the problem in two different ways:

- Tap **Apply Migrations** and, once the page updates, refresh the page; or
- Run the following from the command line in the project's directory:

```
dotnet ef database update
```

The app displays the email used to register the new user and a **Log off** link.
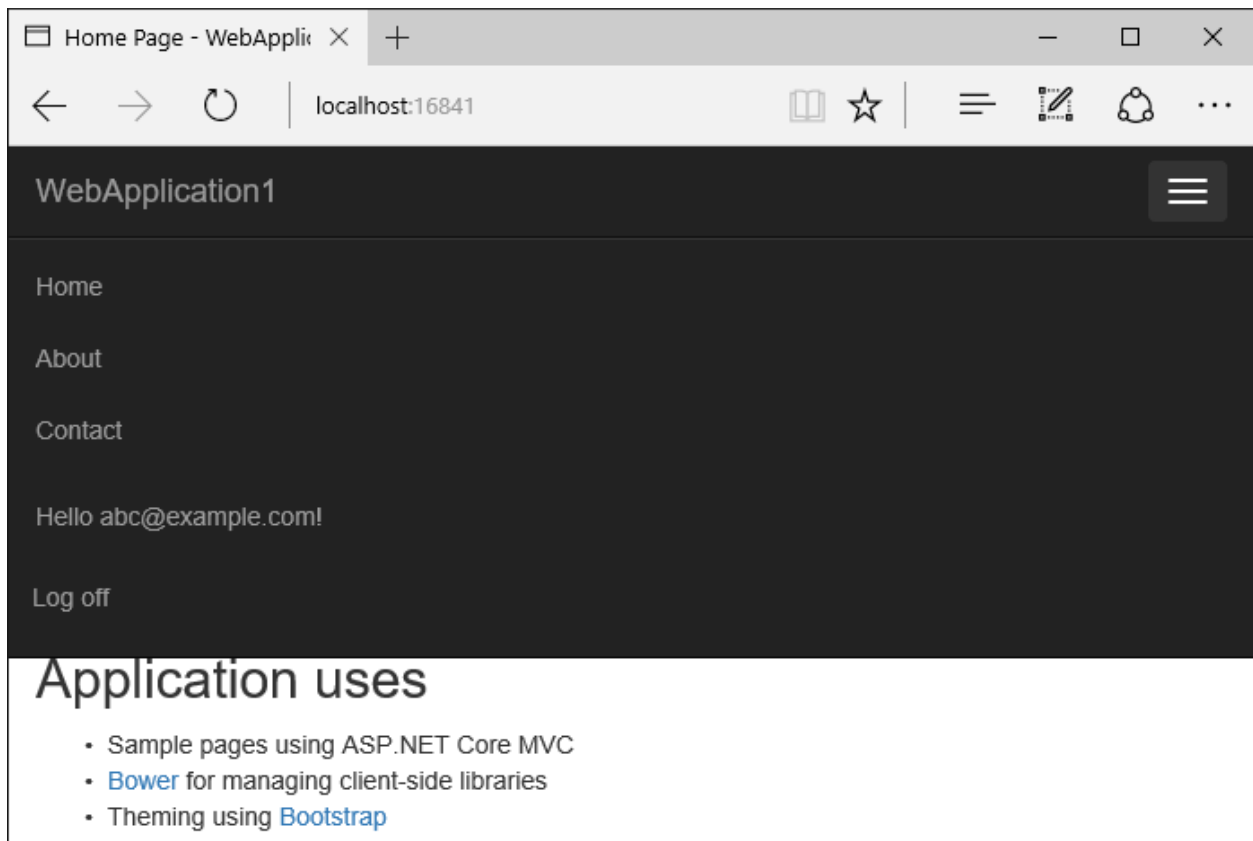
Deploy the app to Azure

Right-click on the project in Solution Explorer and select **Publish...**.
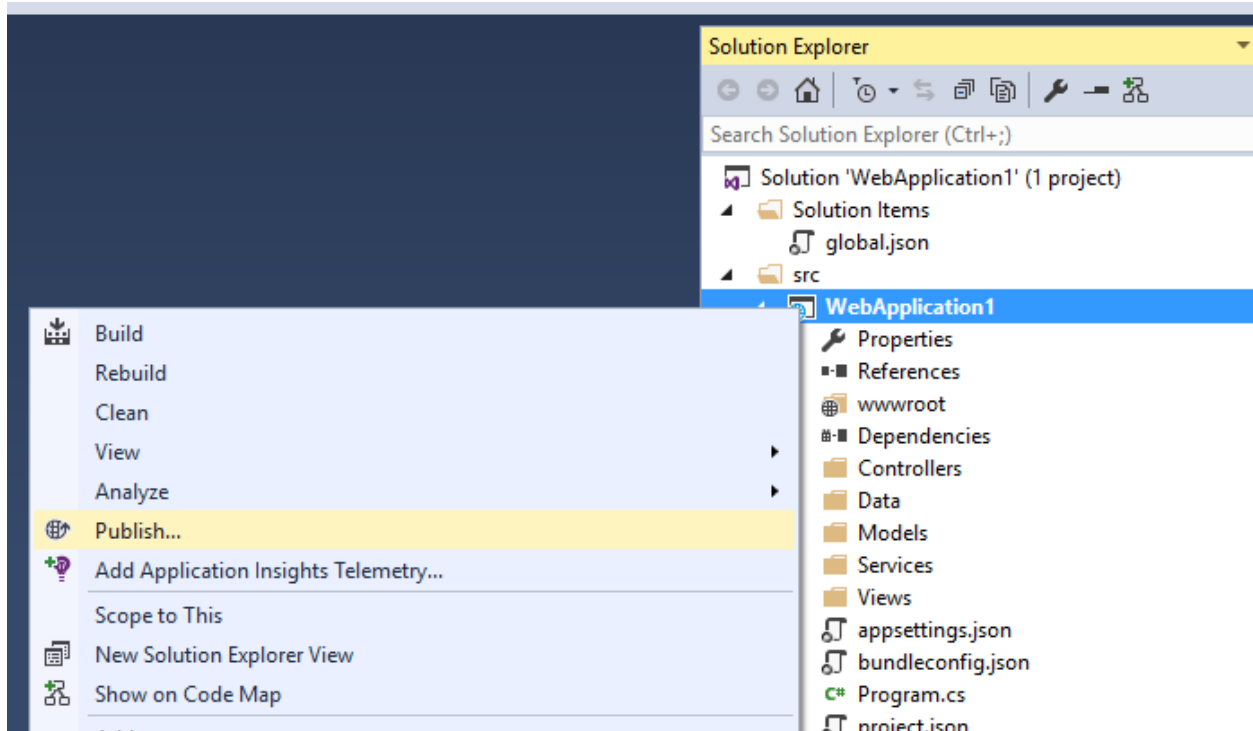
In the **Publish** dialog, tap **Microsoft Azure App Service**.
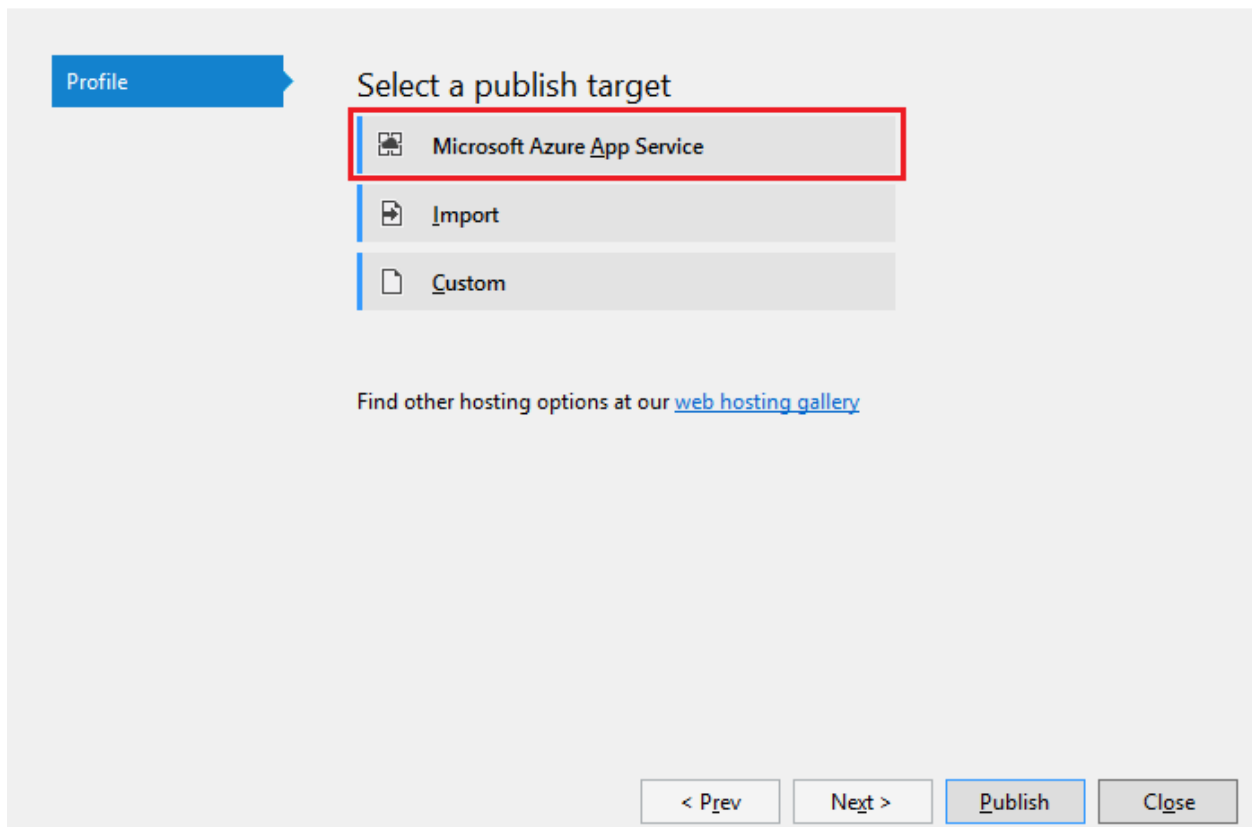
Tap **New...** to create a new resource group. Creating a new resource group will make it easier to delete all the Azure resources you create in this tutorial.

Create a new resource group and app service plan:

- Tap **New...** for the resource group and enter a name for the new resource group
- Tap **New...** for the app service plan and select a location near you. You can keep the default generated name
- Tap **Explore additional Azure services** to create a new database
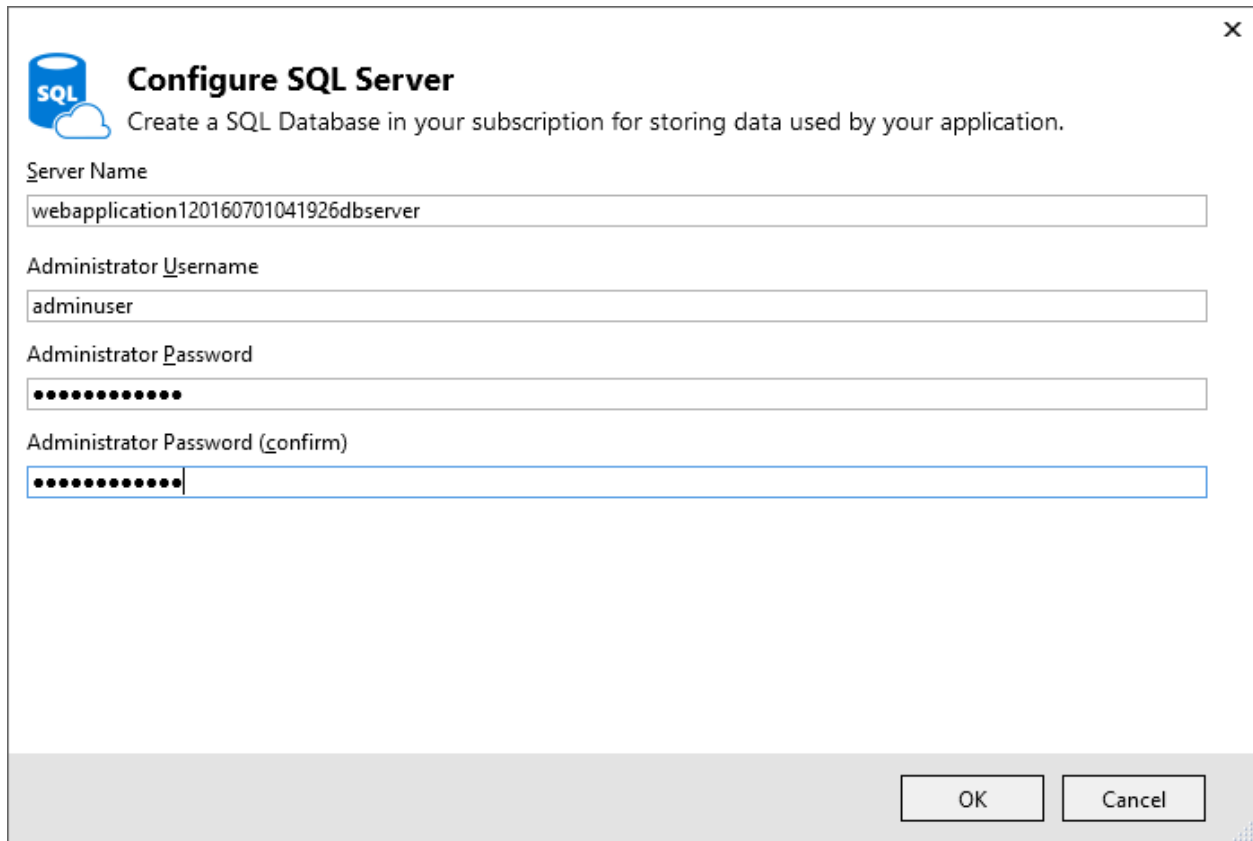
- Tap the green **+** icon to create a new SQL Database
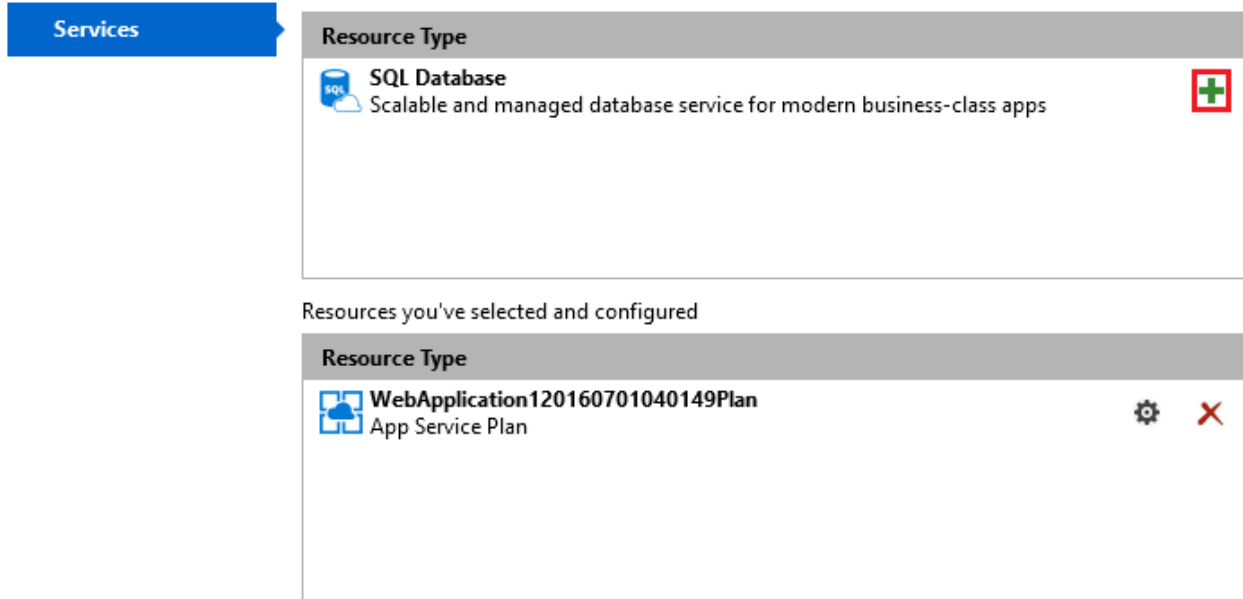
- Tap **New...** on the **Configure SQL Database** dialog to create a new database server.
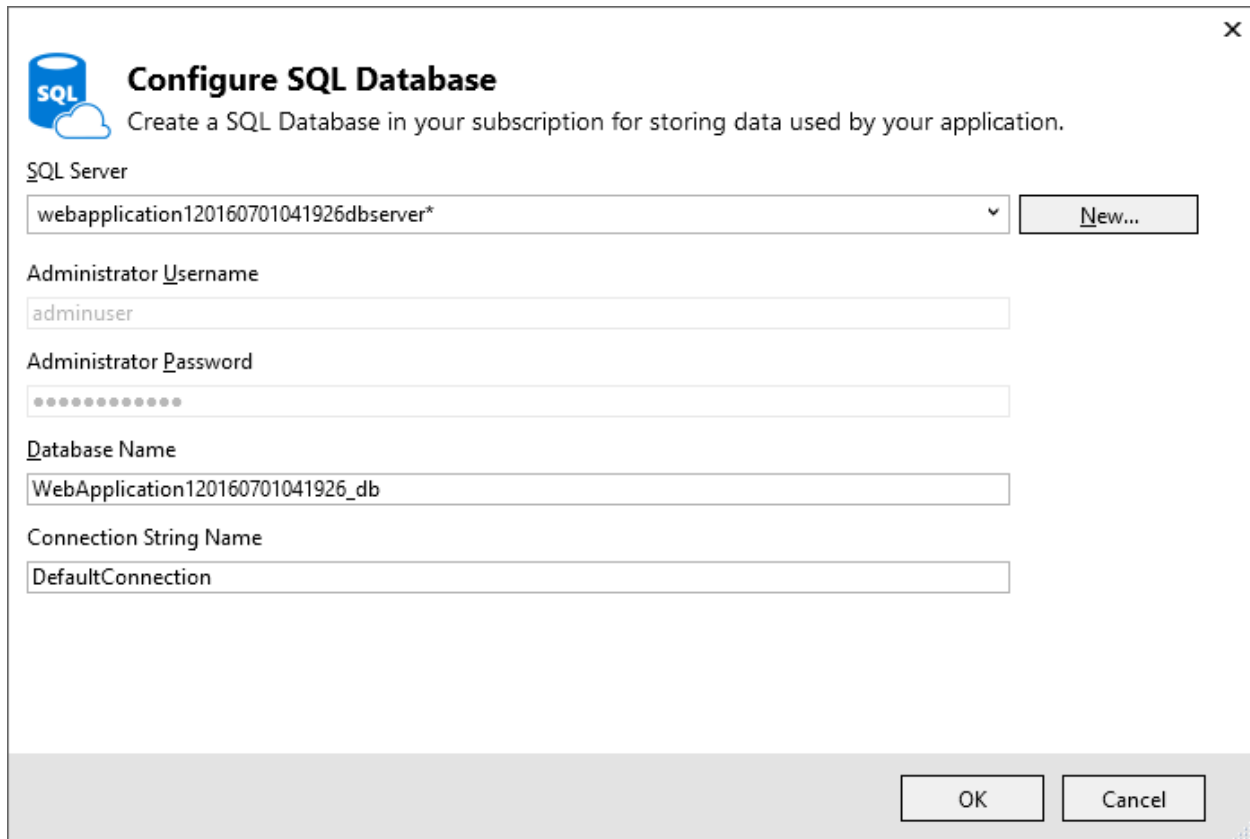
- Enter an administrator user name and password, and then tap **OK**. Don't forget the user name and password you create in this step. You can keep the default **Server Name**

: "admin" is not allowed as the administrator user name.

- Tap **OK** on the **Configure SQL Database** dialog
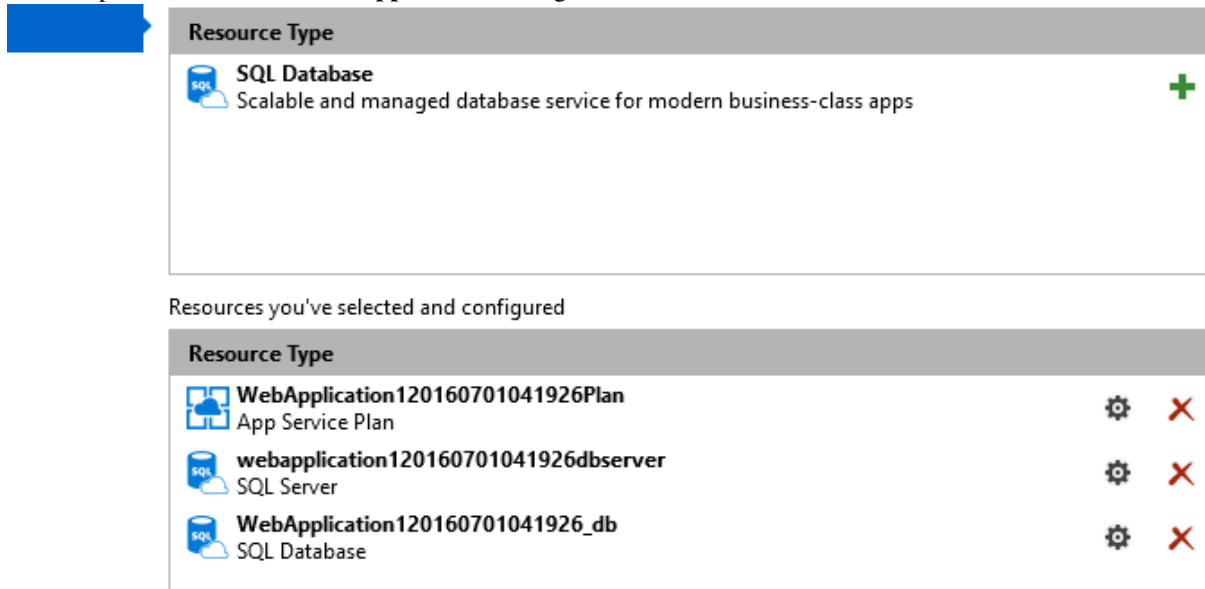
- Tap **Create** on the **Create App Service** dialog

• Tap **Next** in the **Publish** dialog

• On the **Settings** stage of the **Publish** dialog:

  – Expand **Databases** and check **Use this connection string at runtime**

  – Expand **Entity Framework Migrations** and check **Apply this migration on publish**

• Tap **Publish** and wait until Visual Studio finishes publishing your app
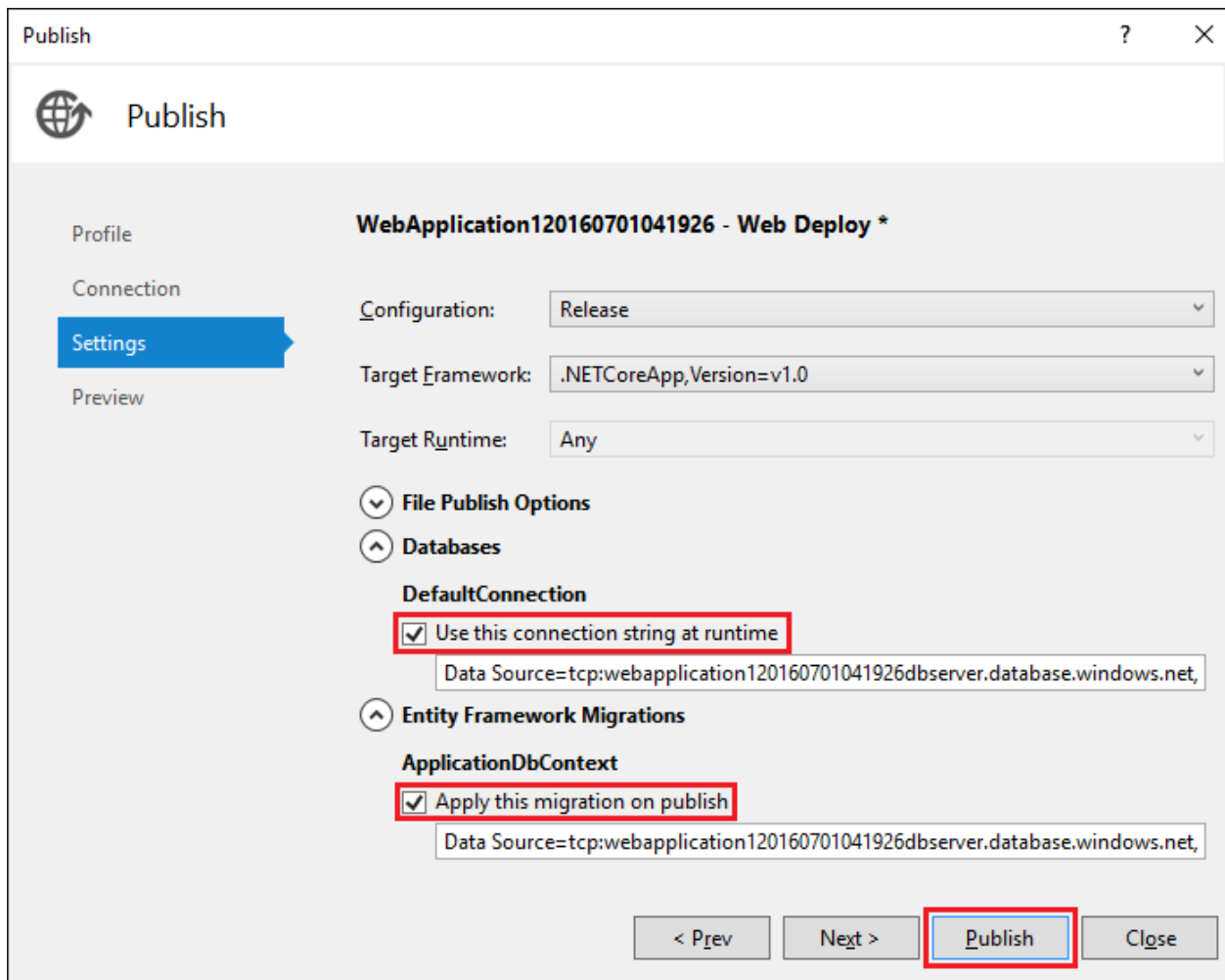
Visual Studio will publish your app to Azure and launch the cloud app in your browser.

Test your app in Azure

- Test the **About** and **Contact** links
- Register a new user
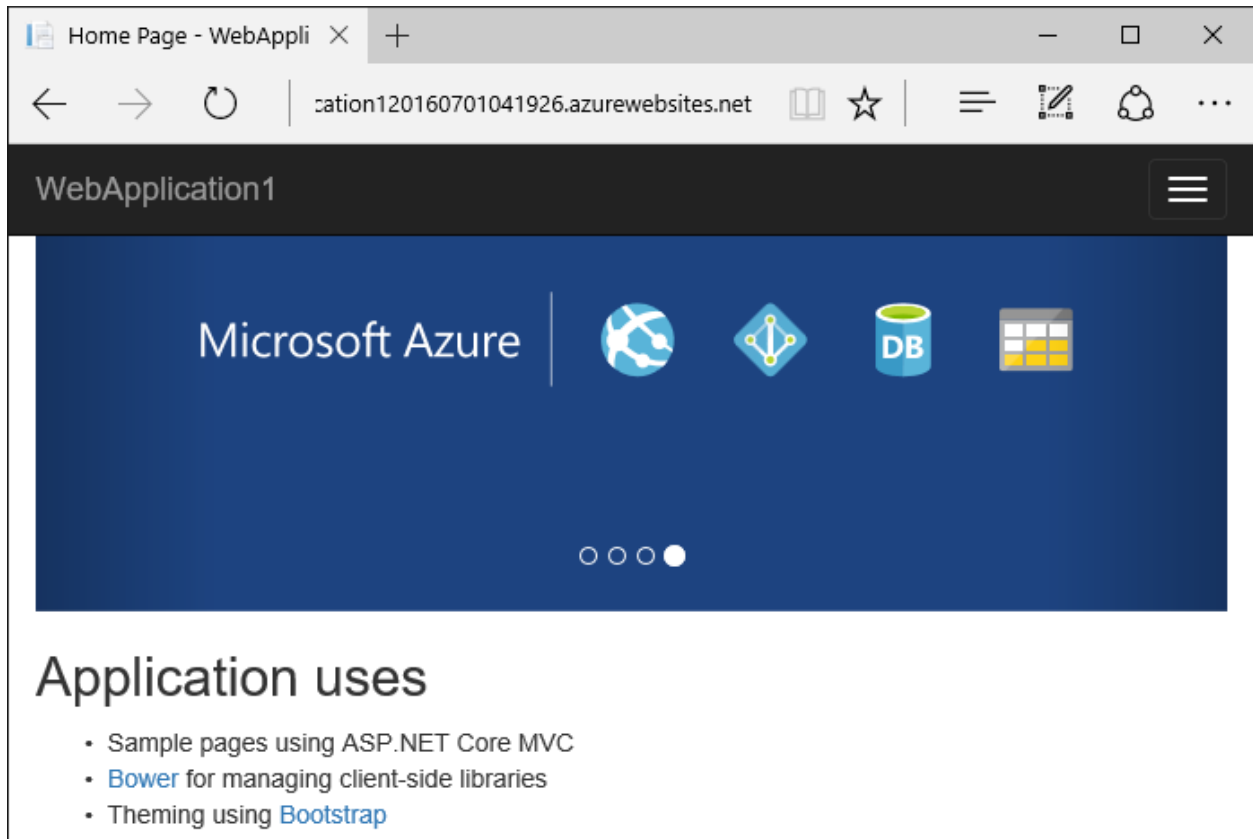
Update the app

- Edit the `Views/Home/About.cshtml` Razor view file and change its contents. For example:

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

<p>My updated about page.</p>
```

- Right-click on the project and tap **Publish...** again
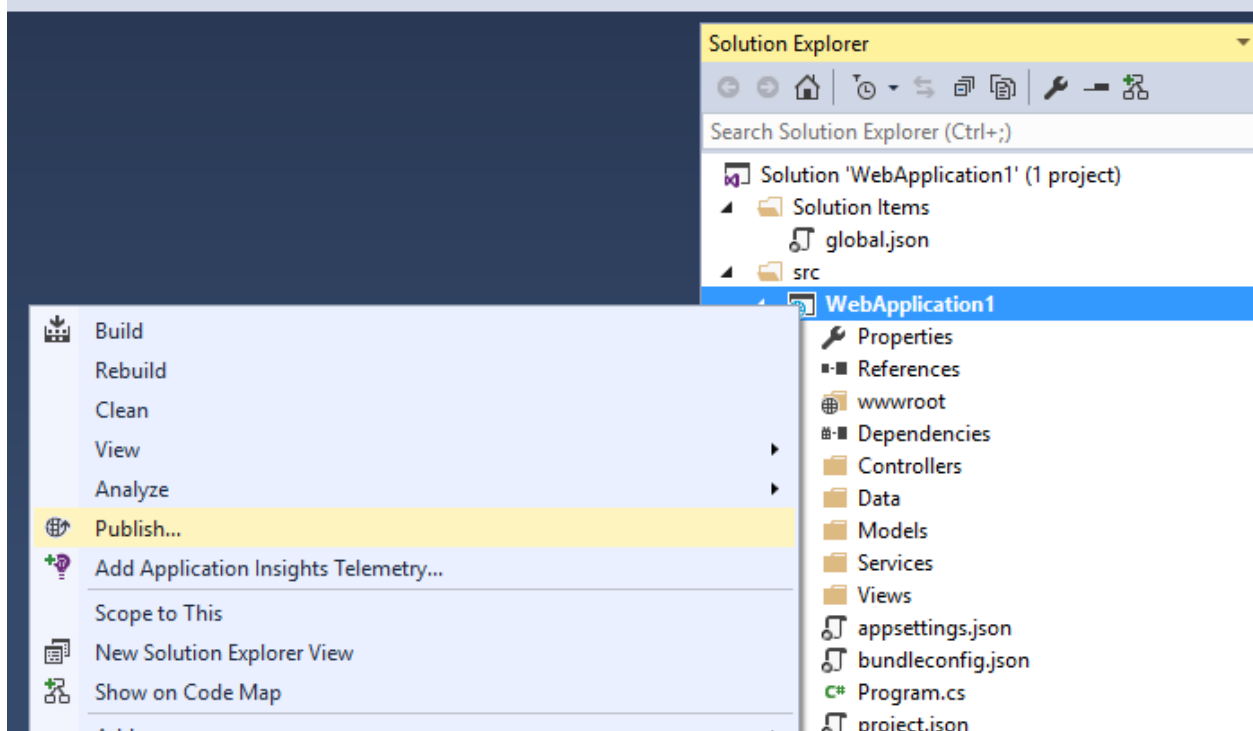
- After the app is published, verify the changes you made are available on Azure

Clean up

When you have finished testing the app, go to the Azure portal and delete the app.

- Select **Resource groups**, then tap the resource group you created
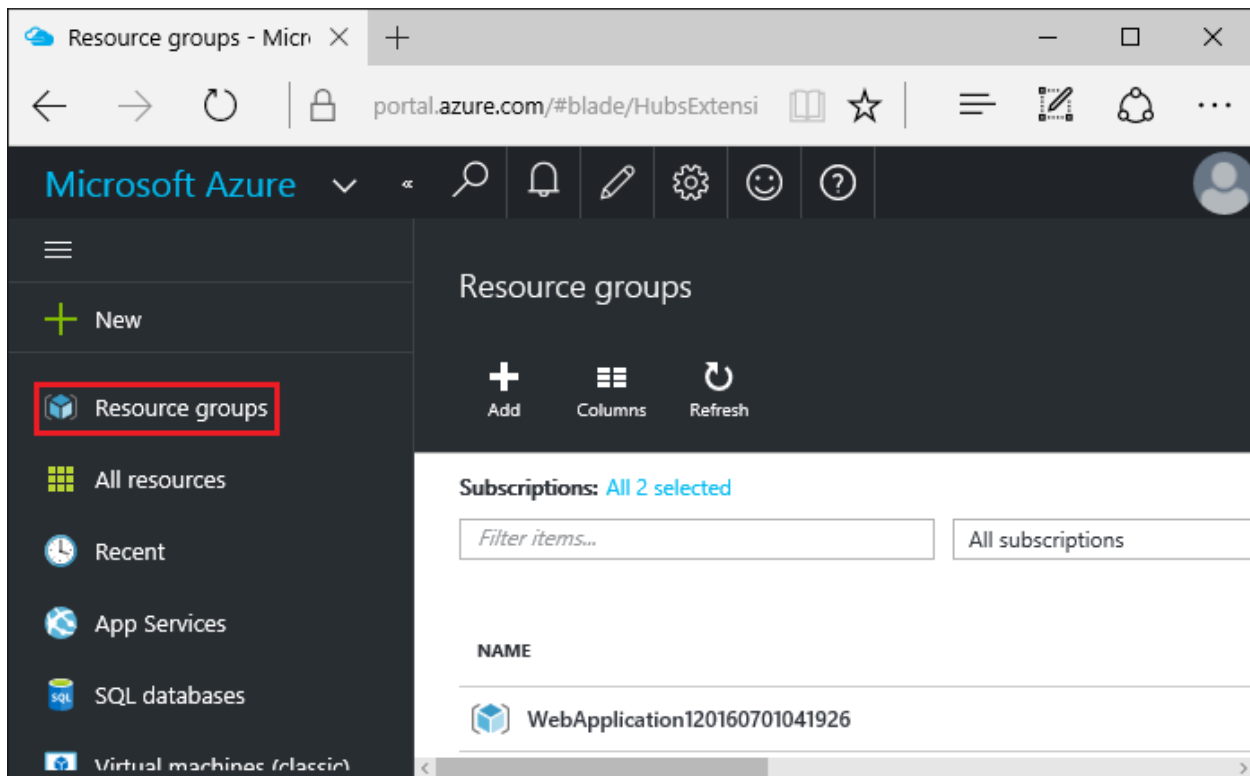
- In the **Resource group** blade, tap **Delete**
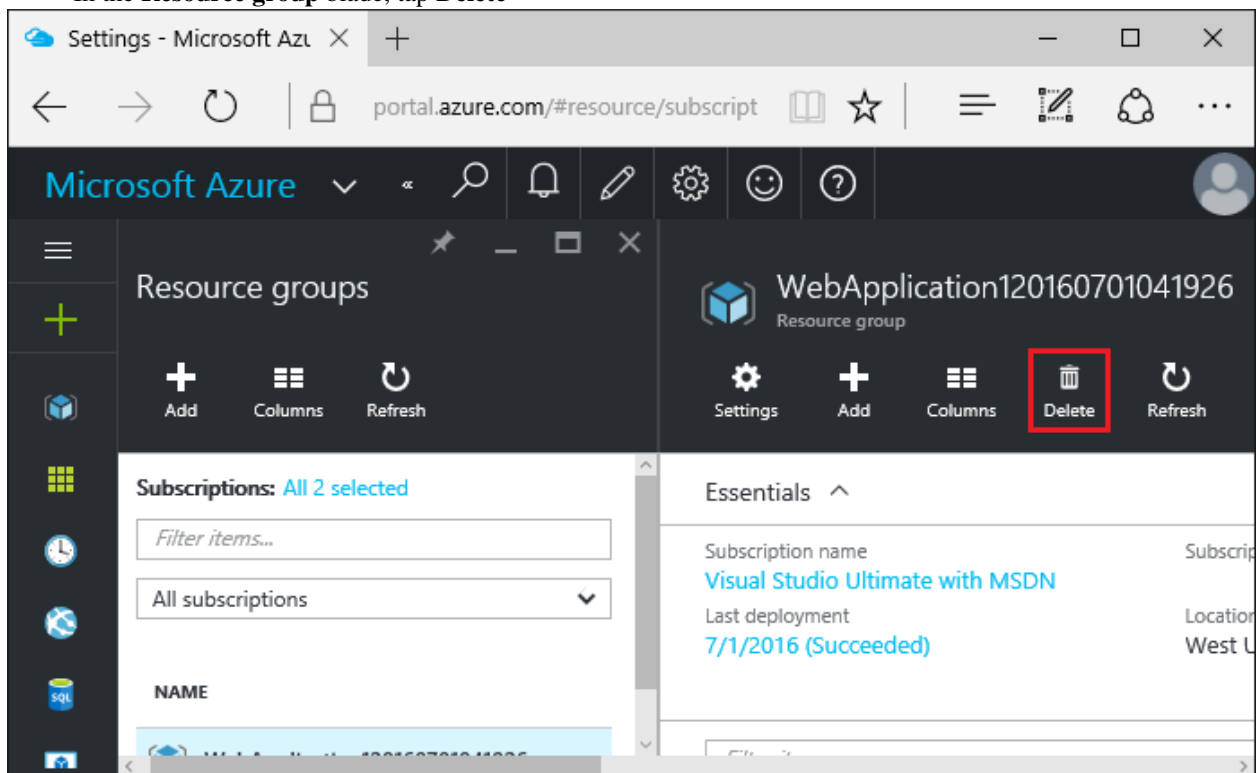
- Enter the name of the resource group and tap **Delete**. Your app and all other resources created in this tutorial are now deleted from Azure

Next steps

- *Getting started with ASP.NET Core MVC and Visual Studio*
- *Introduction to ASP.NET Core*
- *Fundamentals*

1.3.4 Building your first ASP.NET Core MVC app with Visual Studio

Getting started with ASP.NET Core MVC and Visual Studio

By Rick Anderson

This tutorial will teach you the basics of building an ASP.NET Core MVC web app using Visual Studio 2015.

Install Visual Studio and .NET Core

- Install Visual Studio Community 2015. Select the Community download and the default installation. Skip this step if you have Visual Studio 2015 installed.
    - Visual Studio 2015 Home page installer
- Install .NET Core + Visual Studio tooling

Create a web app

From the Visual Studio **Start** page, tap **New Project**.
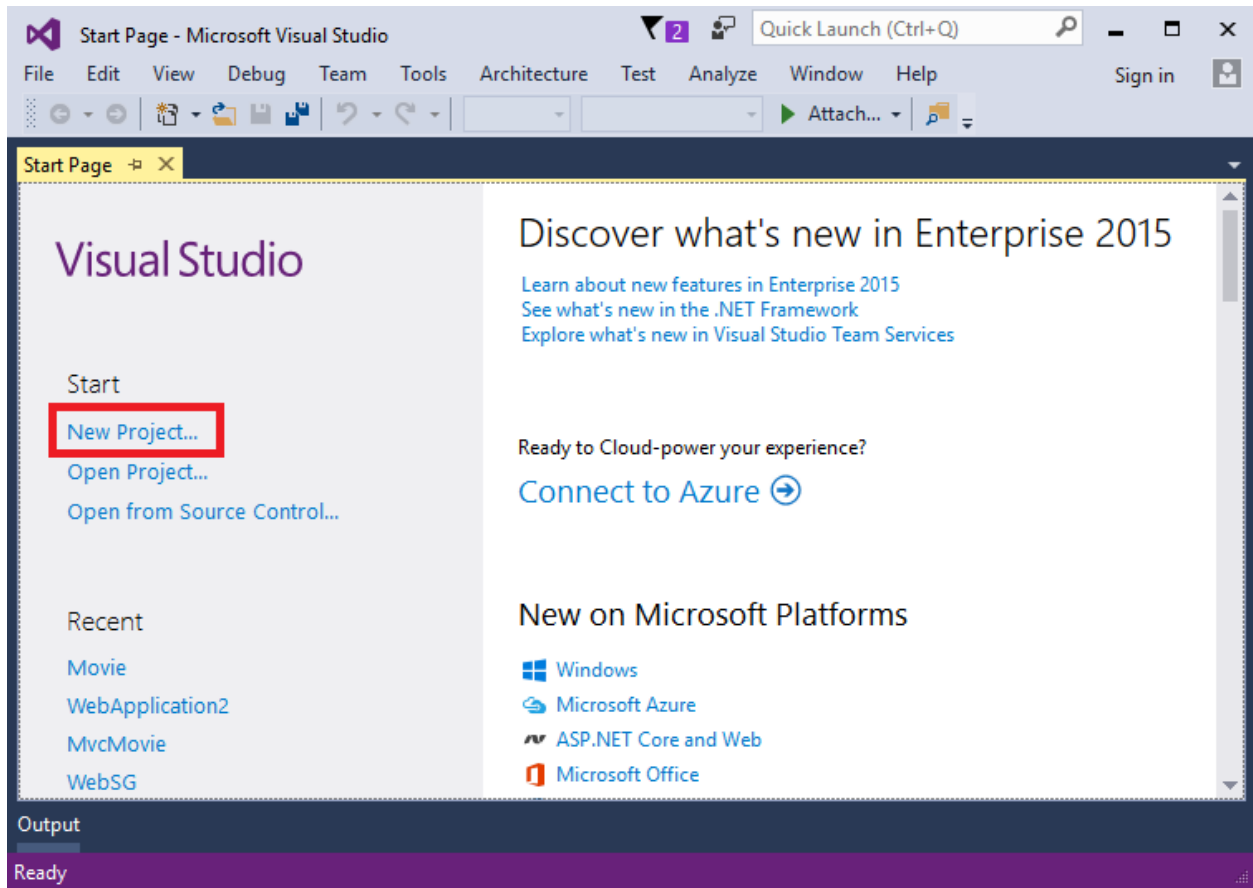
Alternatively, you can use the menus to create a new project. Tap **File > New > Project**.
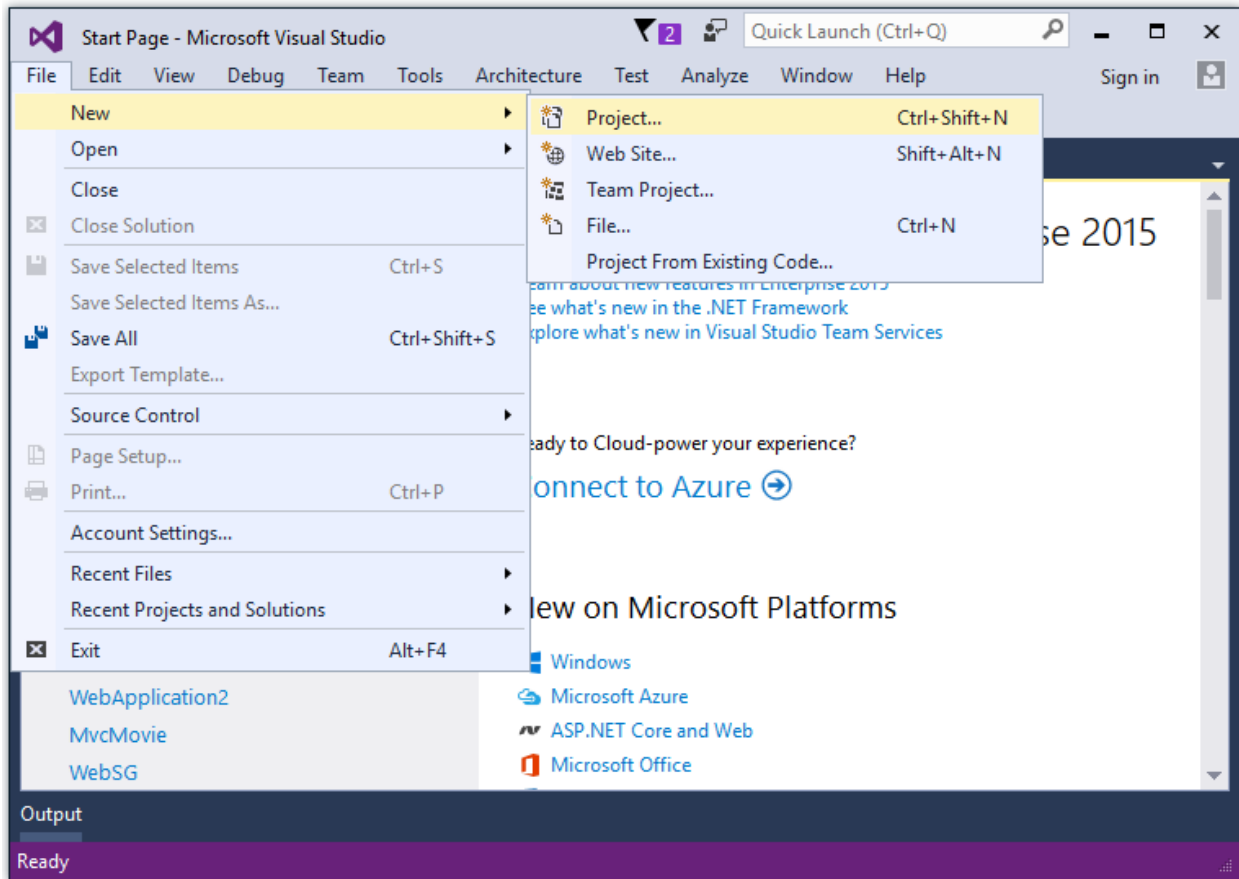
Complete the **New Project** dialog:

- In the left pane, tap **Web**

- In the center pane, tap **ASP.NET Core Web Application (.NET Core)**

- Name the project "MvcMovie" (It's important to name the project "MvcMovie" so when you copy code, the namespace will match.)
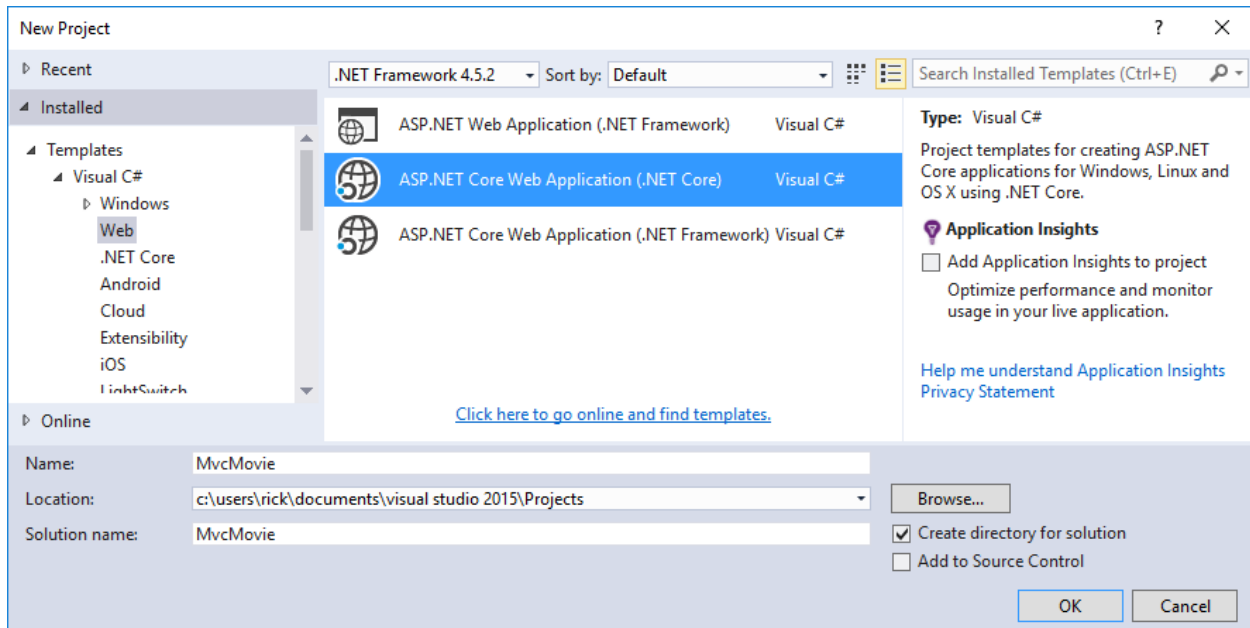
- Tap **OK**

In the **New ASP.NET Core Web Application - MvcMovie** dialog, tap **Web Application**, and then tap **OK**.
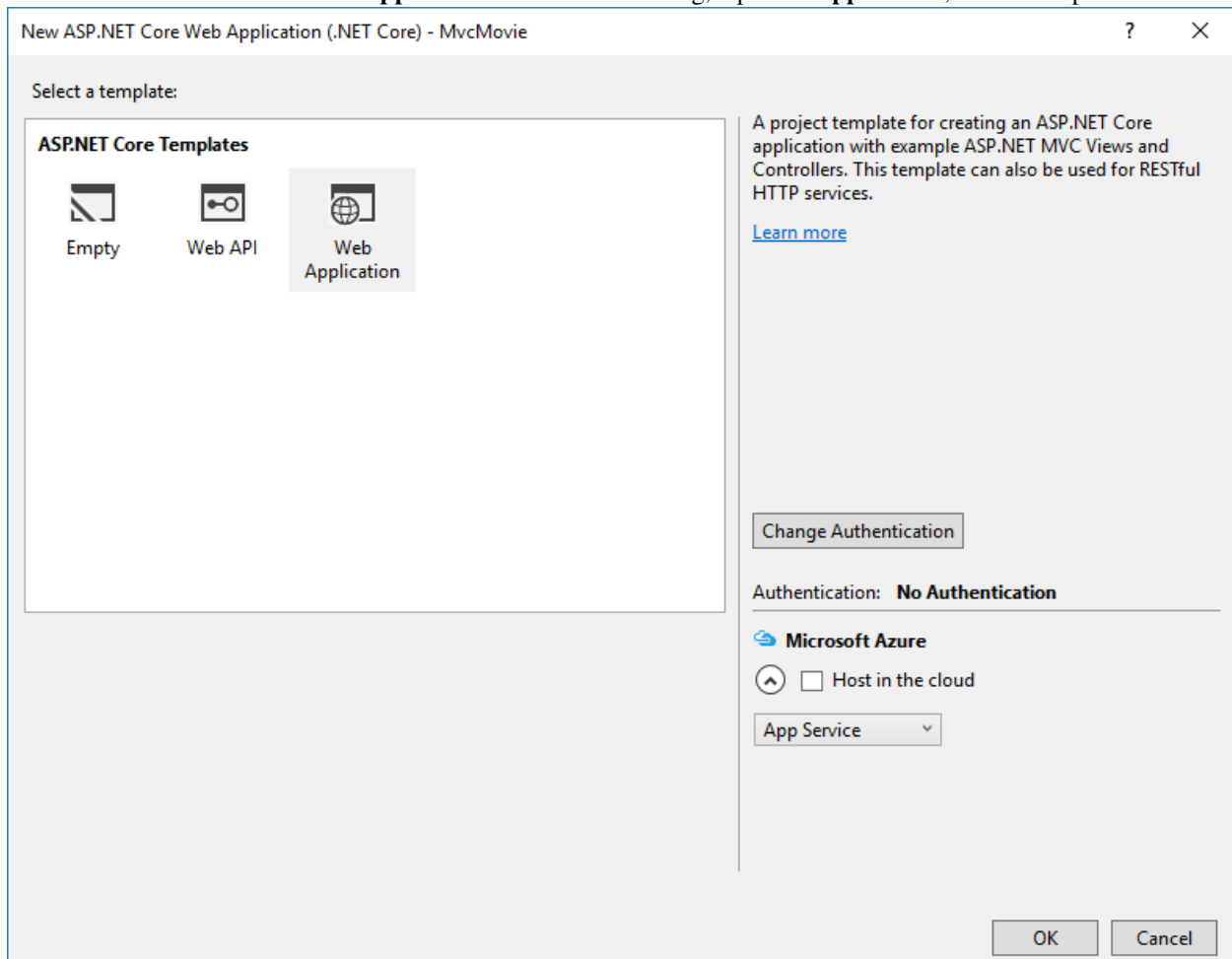
Visual Studio used a default template for the MVC project you just created, so you have a working app right now by

entering a project name and selecting a few options. This is a simple "Hello World!" project, and it's a good place to start,

Tap **F5** to run the app in debug mode or **Ctl-F5** in non-debug mode.
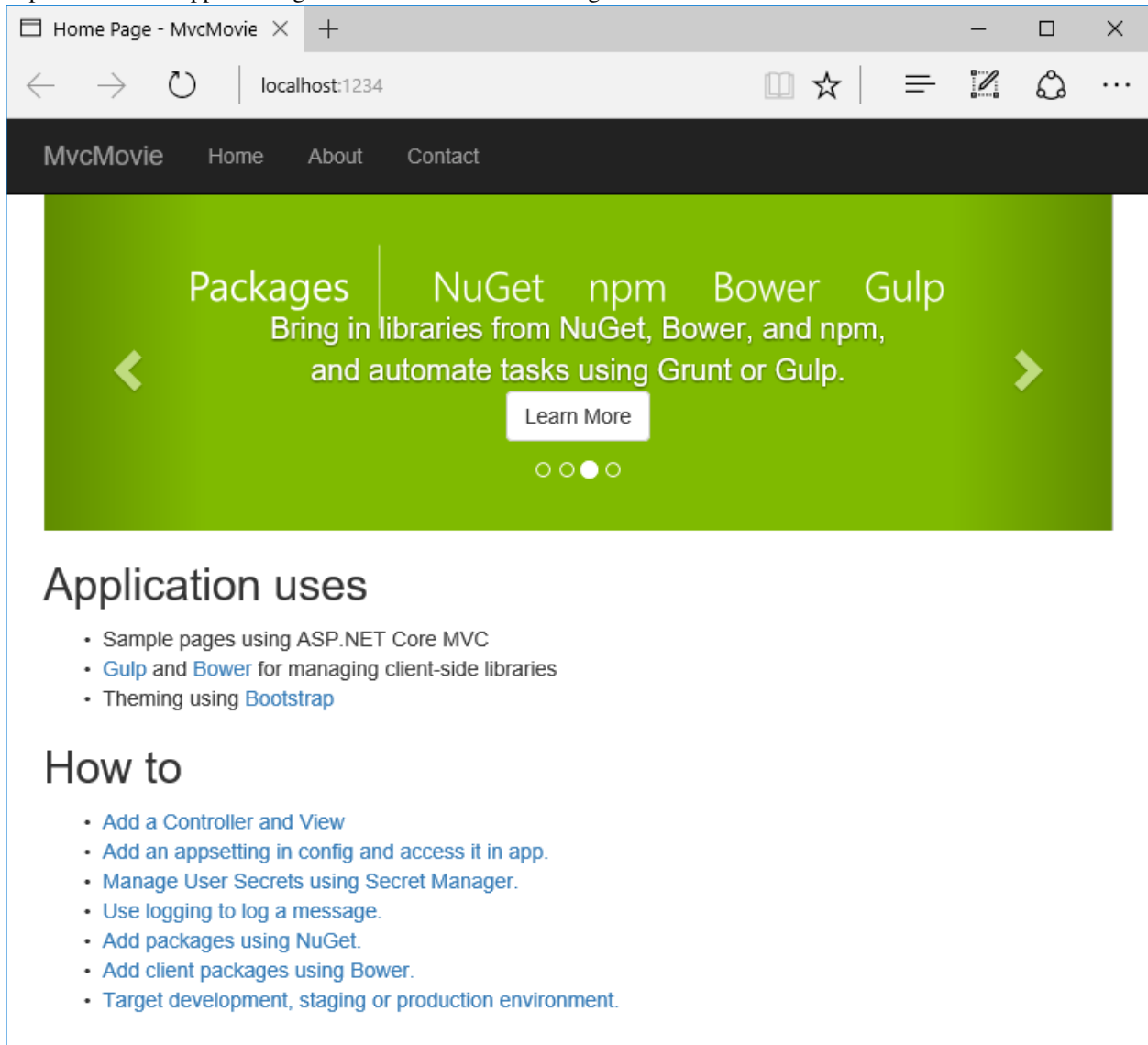
- Visual Studio starts IIS Express and runs your app. Notice that the address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` always points to your own local computer, which in this case is running the app you just created. When Visual Studio creates a web project, a random port is used for the web server. In the image above, the port number is 1234. When you run the app, you'll see a different port number.

- Launching the app with **Ctrl+F5** (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to quickly launch the app and view changes.

- You can launch the app in debug or non-debug mode from the **Debug** menu item:
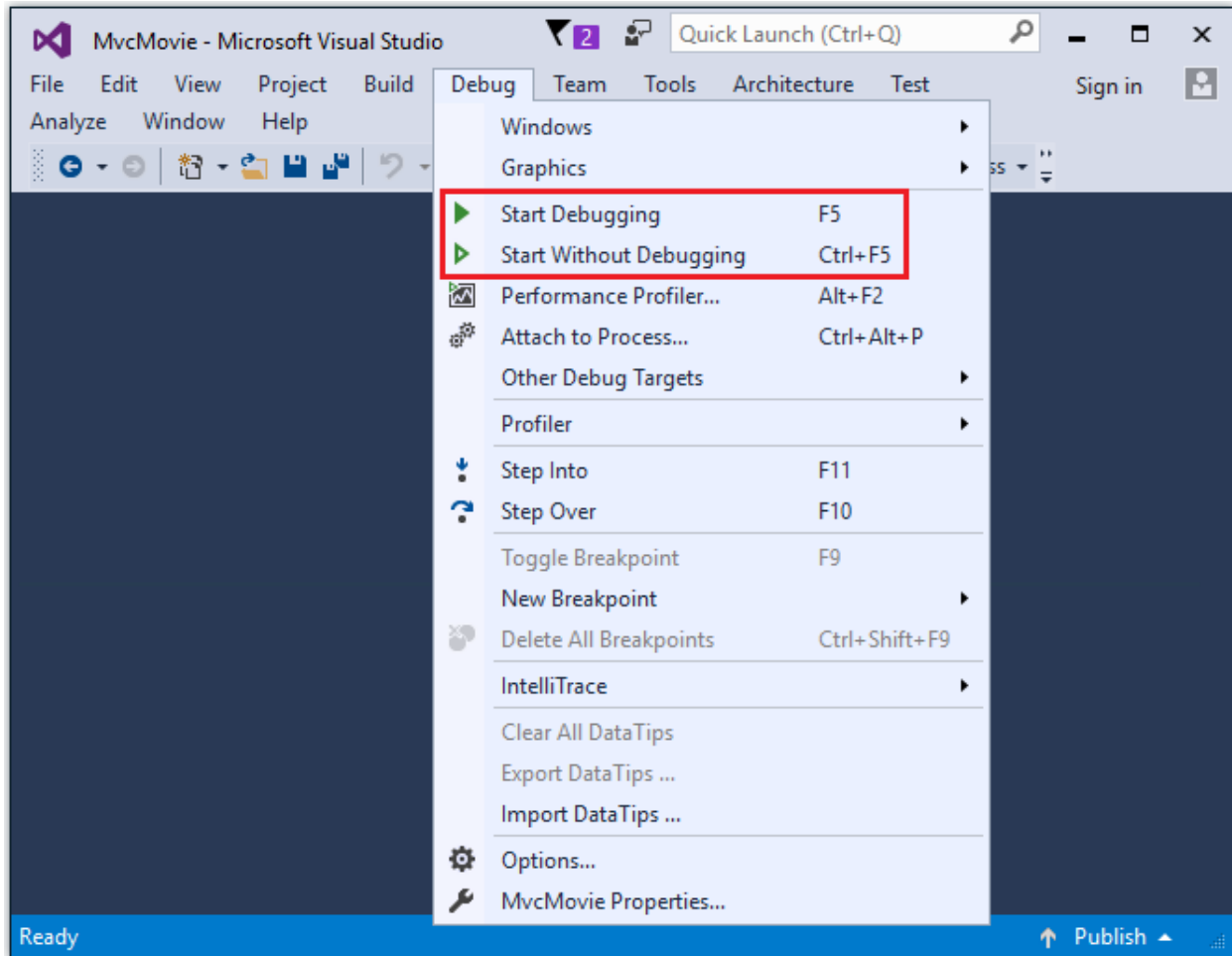
- You can debug the app by tapping the **IIS Express** button
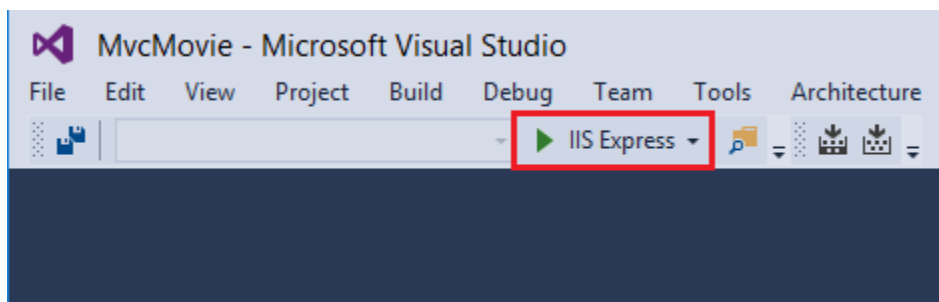
The default template gives you working **Home, Contact, About, Register** and **Log in** links. The browser image above doesn't show these links. Depending on the size of your browser, you might need to click the navigation icon to show them.
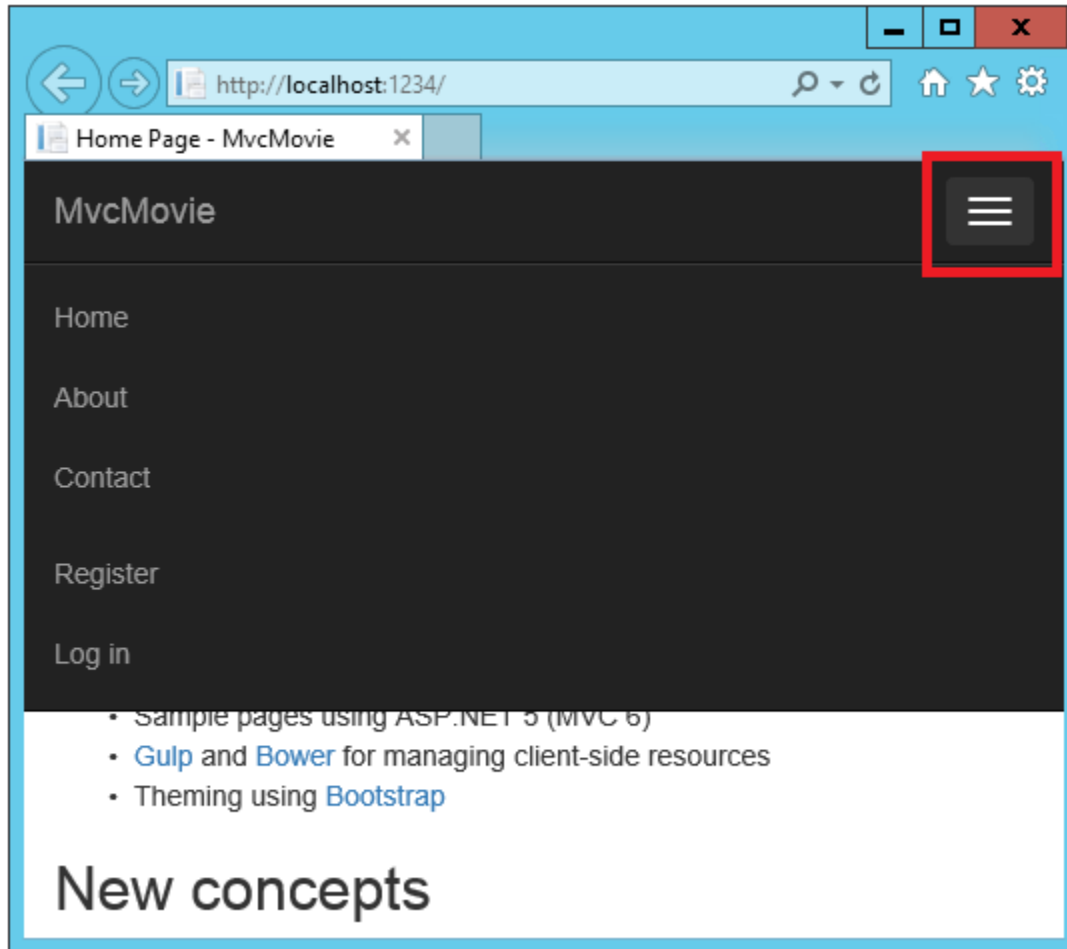
In the next part of this tutorial, we'll learn about MVC and start writing some code.

Adding a controller

By Rick Anderson

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

- **M**odels: Classes that represent the data of the app and that use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a SQL Server database.

- **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.

- **C**ontrollers: Classes that handle browser requests, retrieve model data, and then specify view templates that return a response to the browser. In an MVC app, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles route data and query-string values, and passes these values to the model. The model might use these values to query the database.

The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic

should be located in the app. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

We'll be covering all these concepts in this tutorial series and show you how to use them to build a simple movie app. The following image shows the *Models*, *Views* and *Controllers* folders in the MVC project.
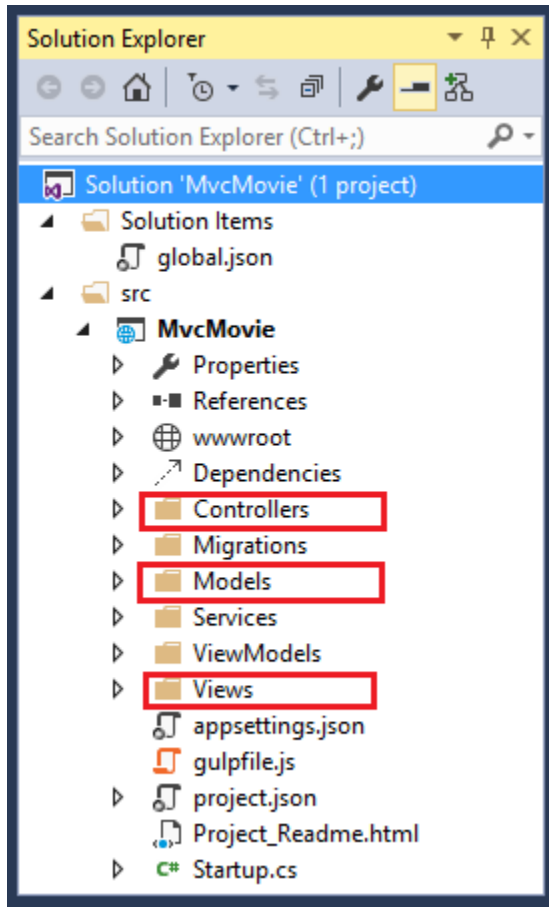
- In **Solution Explorer**, right-click **Controllers > Add > New Item...**

- In the **Add New Item** dialog, enter **HelloWorldController**.

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

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }
```

```
        //
        // GET: /HelloWorld/Welcome/

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

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

```
public class HelloWorldController : Controller
{
    //
    // GET: /HelloWorld/

    public string Index()
    {
        return "This is my default action...";
    }

    //
    // GET: /HelloWorld/Welcome/

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

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

Run the app in non-debug mode (press Ctrl+F5) and append "HelloWorld" to the path in the address bar. (In the image below, http://localhost:1234/HelloWorld is used, but you'll have to replace *1234* with the port number of your app.) The `Index` method returns a string. You told the system to return some HTML, and it did!
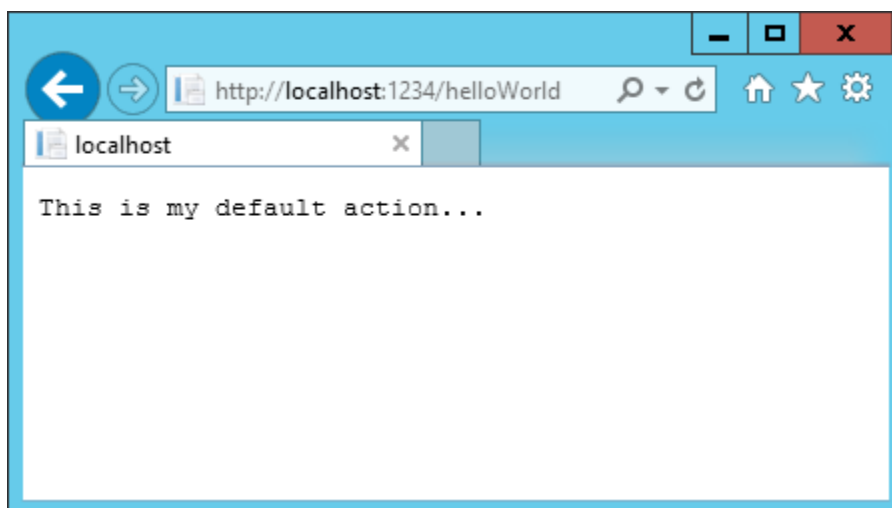
MVC invokes controller classes (and the action methods within them) depending on the incoming URL. The default *URL routing logic* used by MVC uses a format like this to determine what code to invoke:

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

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

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

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

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

Browse to `http://localhost:xxxx/HelloWorld/Welcome`. The `Welcome` method runs and returns the string "This is the Welcome action method...". For this URL, the controller is `HelloWorld` and `Welcome` is the action method. We haven't used the `[Parameters]` part of the URL yet.
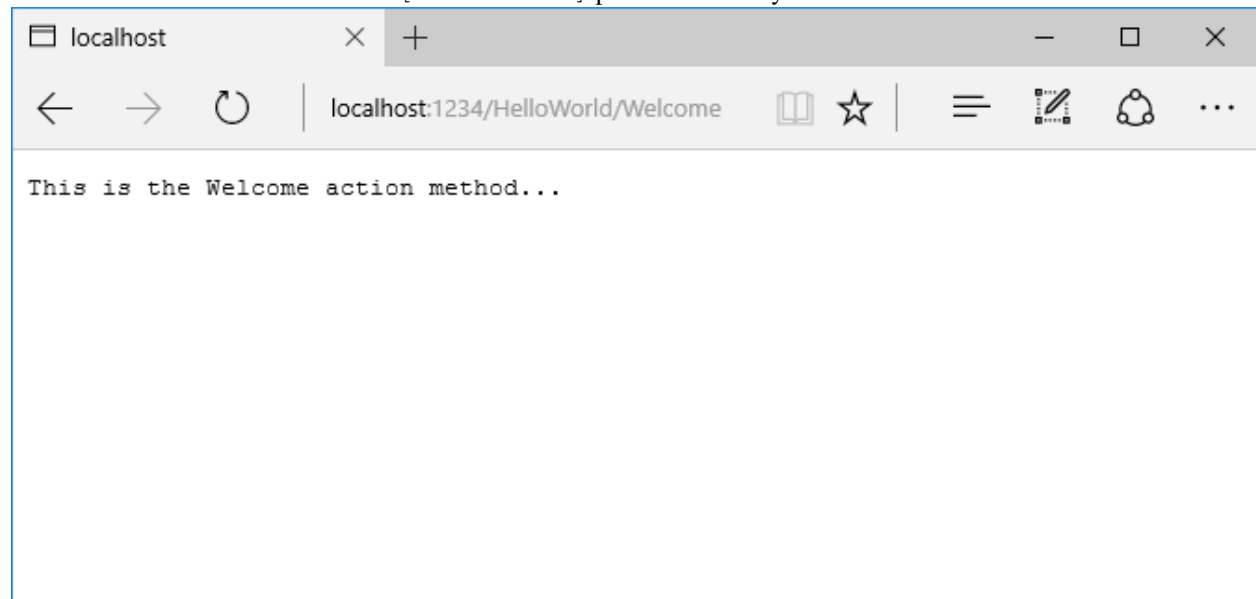
Let's modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, `/HelloWorld/Welcome?name=Scott&numtimes=4`). Change the `Welcome` method to include two parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.

```
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, id: {numTimes}");
}
```

: The code above uses `HtmlEncoder.Default.Encode` to protect the app from malicious input (namely JavaScript). It also uses Interpolated Strings.

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

Run your app and browse to:

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

(Replace xxxx with your port number.) You can try different values for `name` and `numtimes` in the URL. The MVC *model binding* system automatically maps the named parameters from the query string in the address bar to parameters in your method. See *Model Binding* for more information.
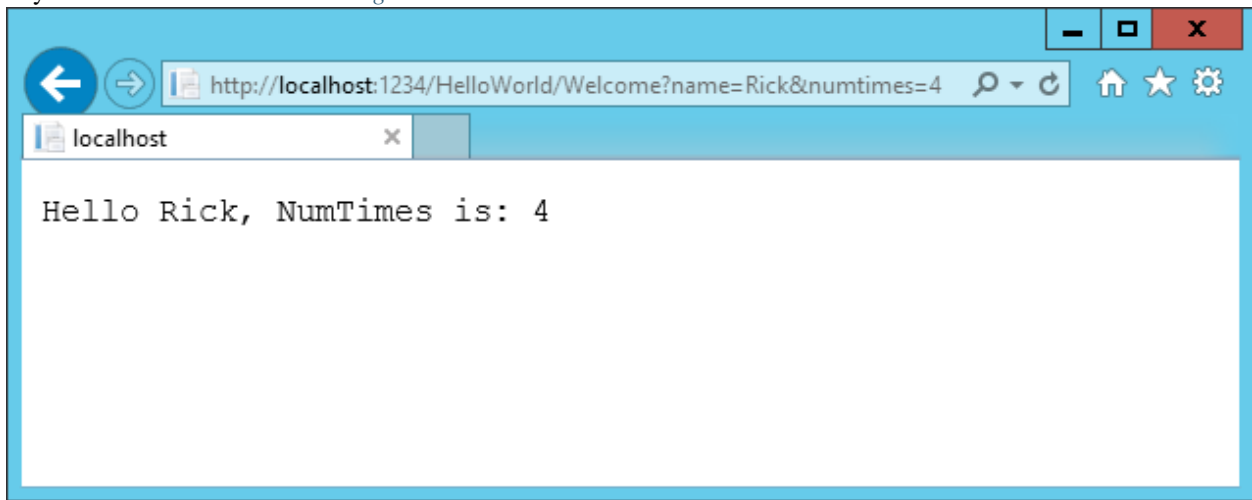
In the sample above, the URL segment (`Parameters`) is not used, the `name` and `numTimes` parameters are passed as query strings. The `?` (question mark) in the above URL is a separator, and the query strings follow. The `&` character separates query strings.

Replace the `Welcome` method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, id: {ID}");
}
```

Run the app and enter the following URL: `http://localhost:xxx/HelloWorld/Welcome/3?name=Rick`
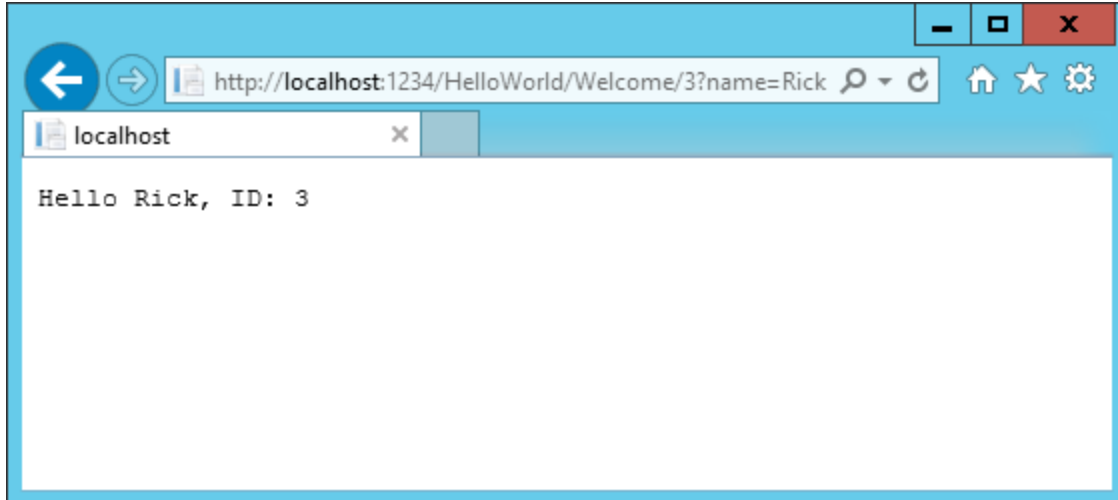
This time the third URL segment matched the route parameter `id`. The `Welcome` method contains a parameter `id` that matched the URL template in the `MapRoute` method. The trailing `?` (in `id?`) indicates the `id` parameter is optional.

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

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

Adding a view

By Rick Anderson

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

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

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

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

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

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

- Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.
- In the **Add New Item - MvcMovie** dialog
  - In the search box in the upper-right, enter *view*
  - Tap **MVC View Page**
  - In the **Name** box, keep the default *Index.cshtml*
  - Tap **Add**
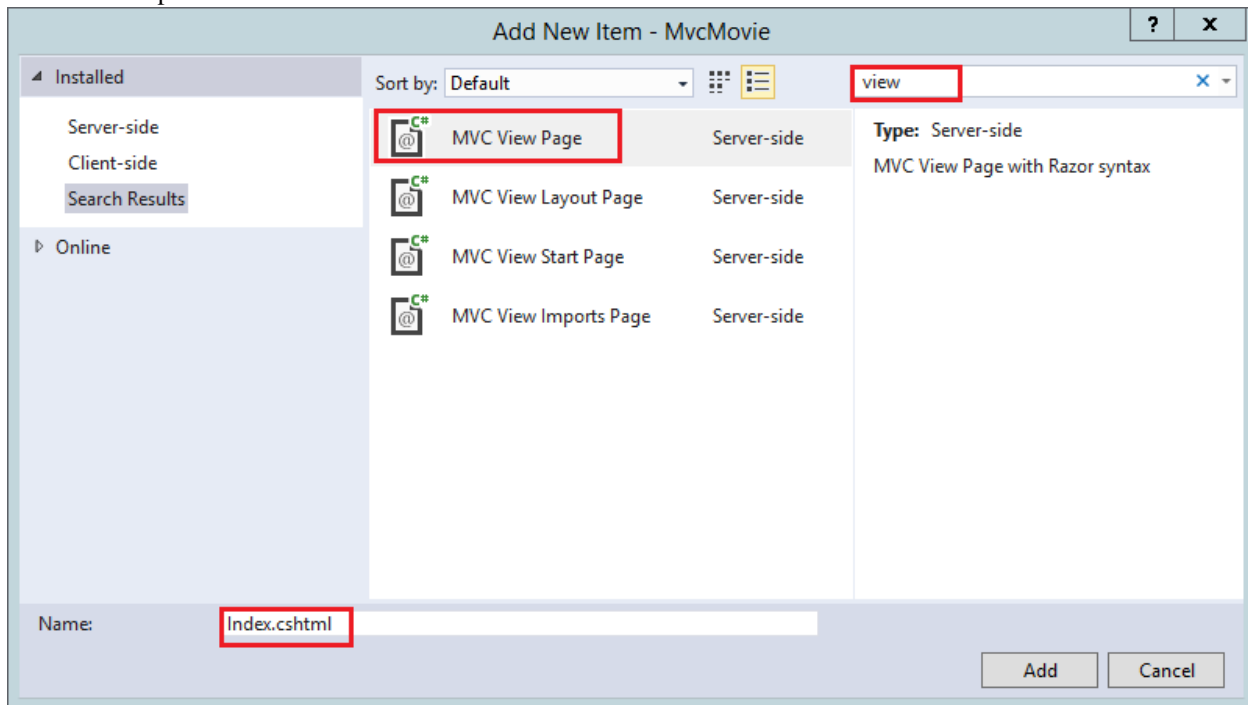
Replace the contents of the *Views/HelloWorld/Index.cshtml* Razor view file with the following:

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

<h2>Index</h2>

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

Navigate to `http://localhost:xxxx/HelloWorld`. The `Index` method in the `HelloWorldController` didn't do much work; it simply ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser. Because you didn't explicitly specify the name of the view template file to use, MVC defaulted to using the *Index.cshtml* view file in the */Views/HelloWorld* folder. The image below shows the string "Hello from our View Template!" hard-coded in the view.
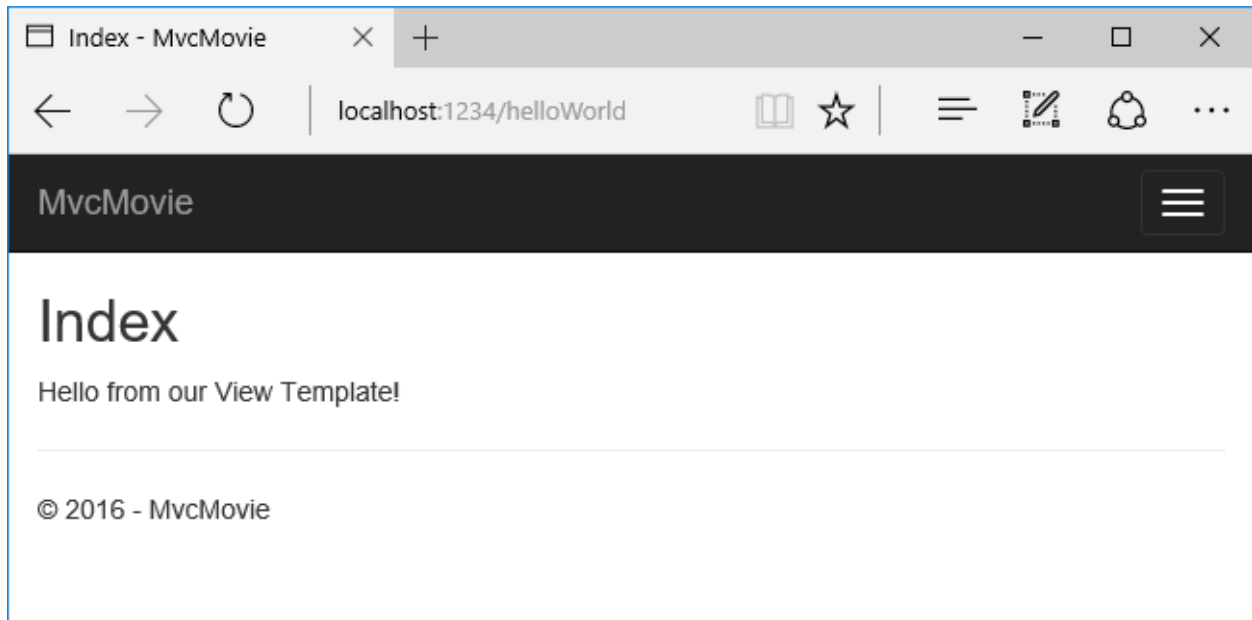
If your browser window is small (for example on a mobile device), you might need to toggle (tap) the Bootstrap navigation button in the upper right to see the to the **Home**, **About**, and **Contact** links.
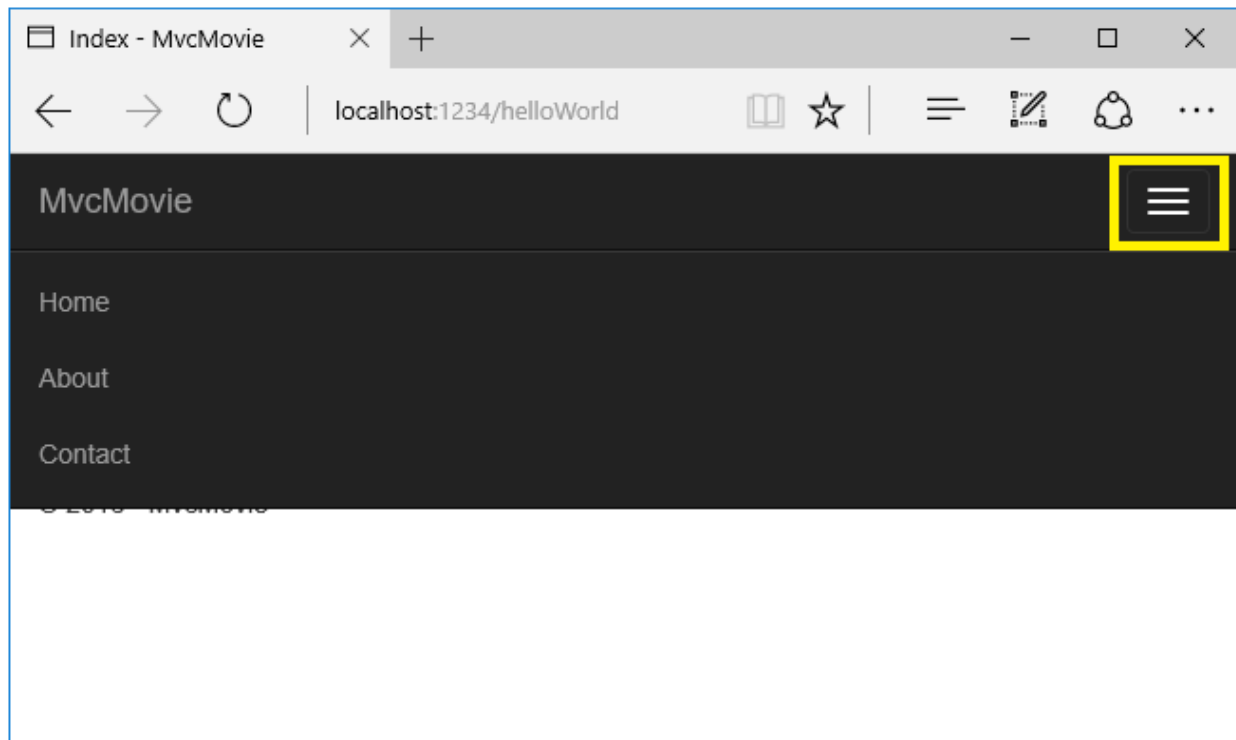
Changing views and layout pages

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

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

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

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App </title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.mi
                asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
                asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-te
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navb
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Movies" asp-action="Index" class="navbar-brand">MvcMov
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
                </ul>
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2016 - MvcMovie</p>
        </footer>
    </div>
```

:   We haven't implemented the `Movies` controller yet, so if you click on that link, you'll get a 404 (Not found) error.

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

Examine the *Views/_ViewStart.cshtml* file:

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

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

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

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

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

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

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

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

`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewDataDictionary` to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

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

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

Also notice how the content in the *Index.cshtml* view template was merged with the *Views/Shared/_Layout.cshtml* view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application. To learn more see *Layout*.
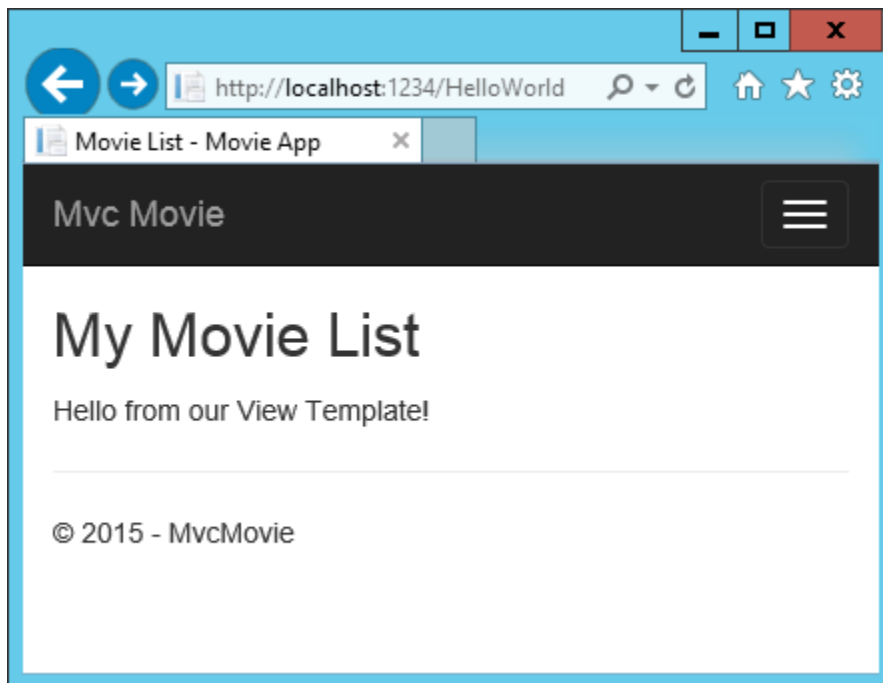
Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet. Shortly, we'll walk through how to create a database and retrieve model data from it.

Passing Data from the Controller to the View

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

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

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

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

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;
```

```
namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

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

- Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.

- In the **Add New Item - MvcMovie** dialog

    - In the search box in the upper-right, enter *view*

    - Tap **MVC View Page**

   - In the **Name** box, enter *Welcome.cshtml*

    - Tap **Add**

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

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

<h2>Welcome</h2>

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

Save your changes and browse to the following URL:

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

Data is taken from the URL and passed to the controller using the *MVC model binder* . The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.
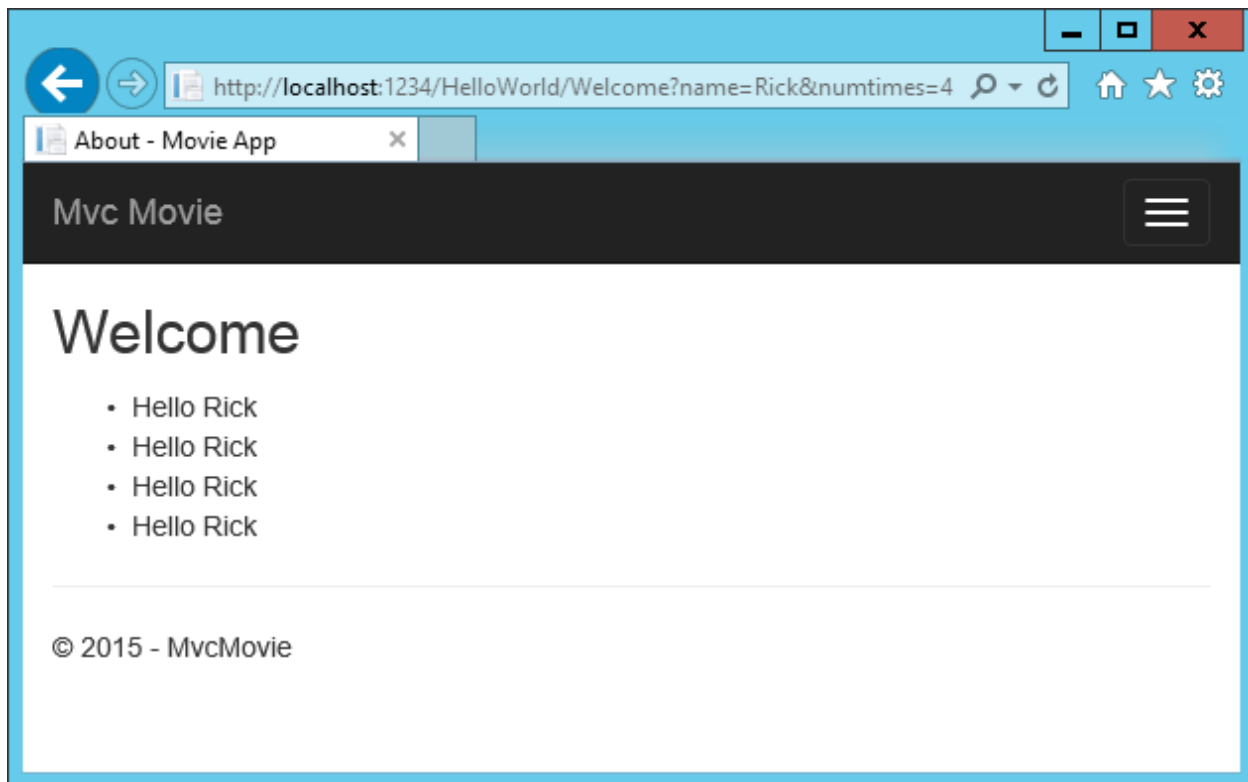
In the sample above, we used the `ViewData` dictionary to pass data from the controller to a view. Later in the tutorial, we will use a view model to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the `ViewData` dictionary approach. See *Dynamic vs Strongly Typed Views* for more information.

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

Adding a model

By Rick Anderson

In this section you'll add some classes for managing movies in a database. These classes will be the "**M**odel" part of the **MVC** app.

You'll use a .NET Framework data-access technology known as the Entity Framework Core to define and work with these data model classes. Entity Framework Core (often referred to as **EF** Core) features a development paradigm called *Code First*. You write the code first, and the database tables are created from this code. Code First allows you to create data model objects by writing simple classes. (These are also known as POCO classes, from "plain-old CLR objects.") The database is created from your classes. If you are required to create the database first, you can still follow this tutorial to learn about MVC and EF app development.

Create a new project with individual user accounts

In the current version of the ASP.NET Core MVC tools for Visual Studio, scaffolding a model is only supported when you create a new project with individual user accounts. We hope to have this fixed in the next tooling update. Until that's fixed, you'll need to create a new project with the same name. Because the project has the same name, you'll need to create it in another directory.

From the Visual Studio **Start** page, tap **New Project**.
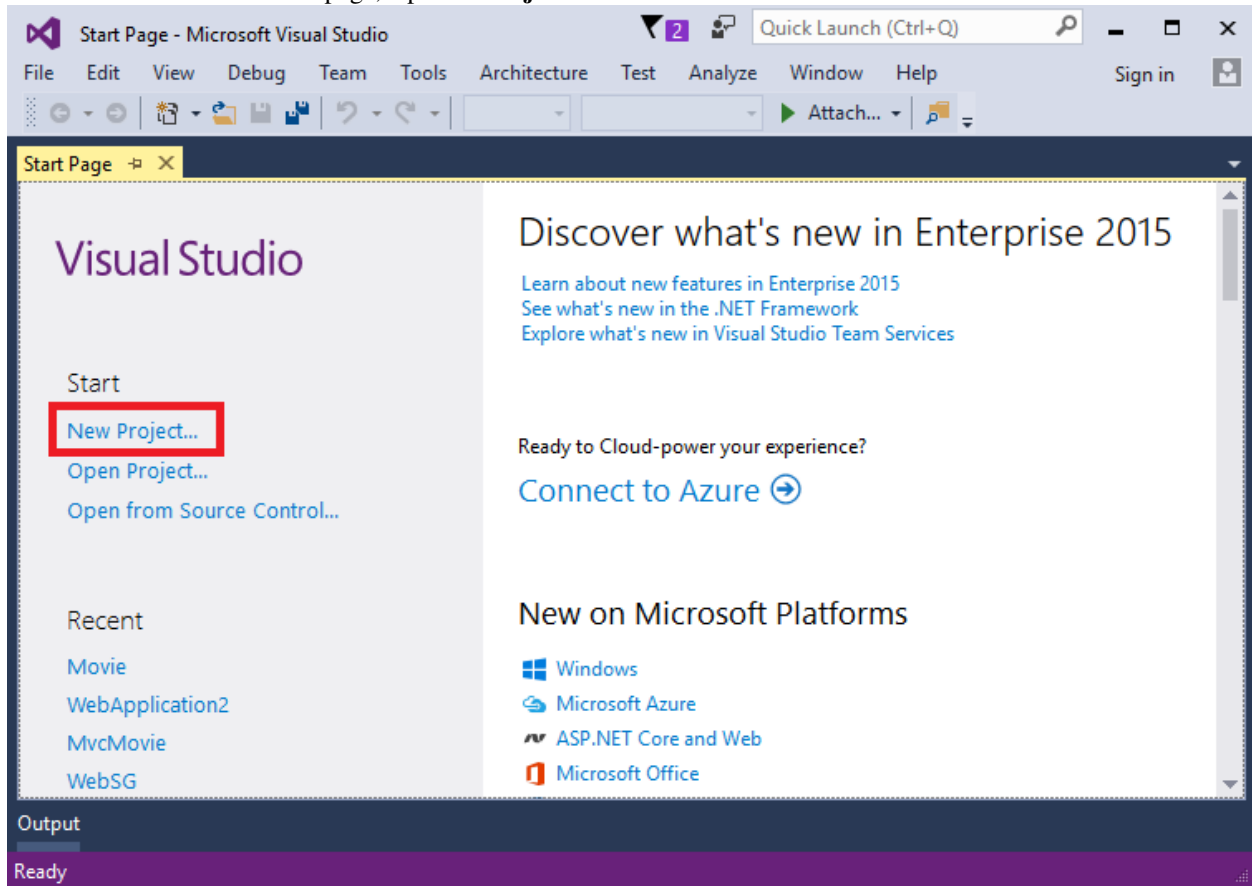
Alternatively, you can use the menus to create a new project. Tap **File > New > Project**.
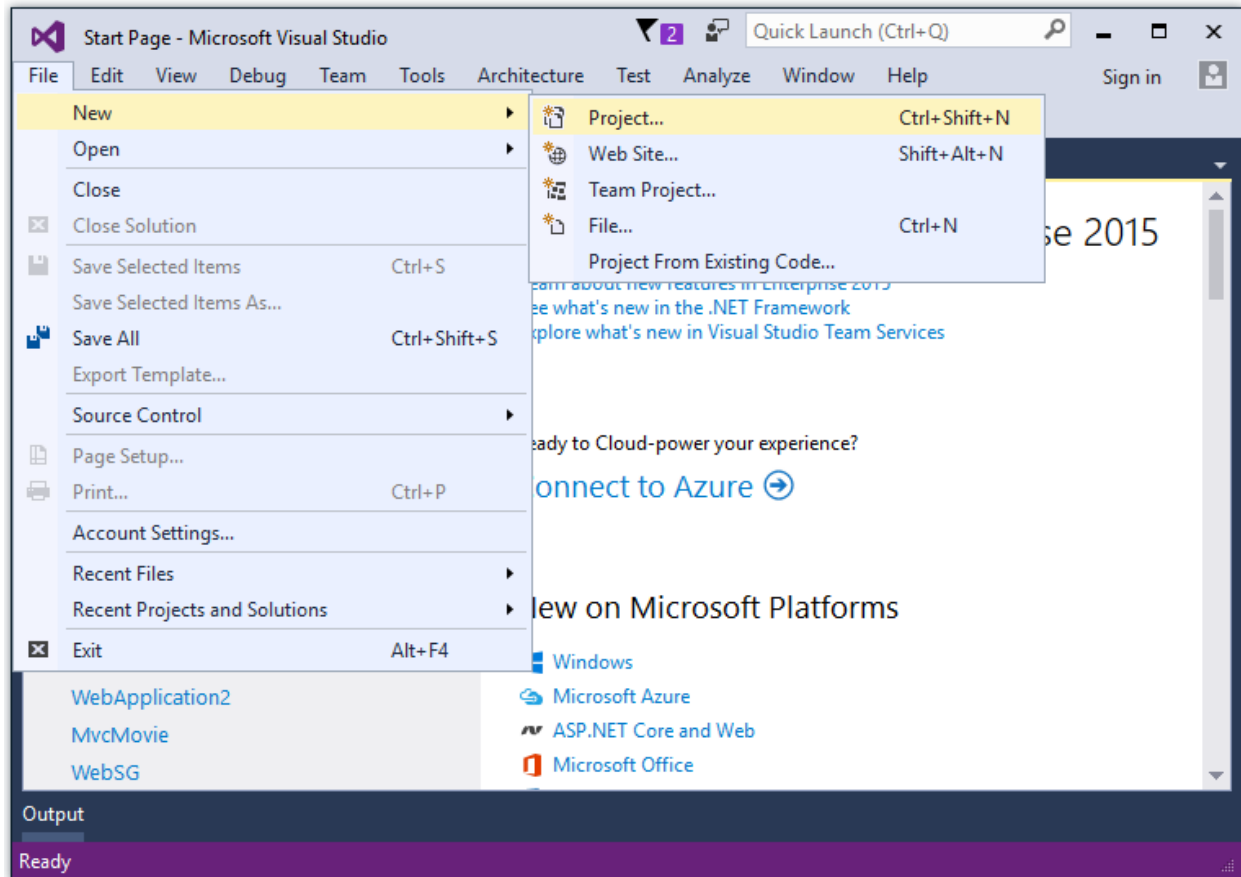
Complete the **New Project** dialog:

- In the left pane, tap **Web**

- In the center pane, tap **ASP.NET Core Web Application (.NET Core)**

- Change the location to a different directory from the previous project you created or you'll get an error

- Name the project "MvcMovie" (It's important to name the project "MvcMovie" so when you copy code, the namespace will match.)

- Tap **OK**
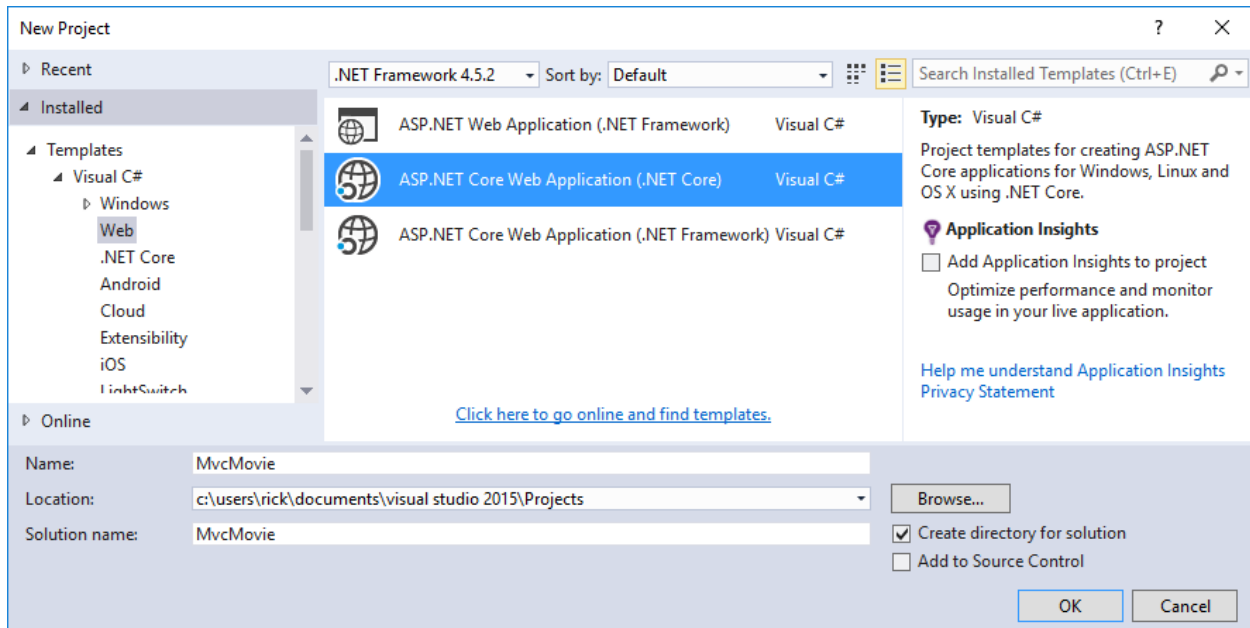
> **:** You must have the **Authentication** set to **Individual User Accounts** in this release for the scaffolding engine to work.

In the **New ASP.NET Core Web Application - MvcMovie** dialog:

- tap **Web Application**
- tap the **Change Authentication** button and change the authentication to **Individual User Accounts** and tap **OK**
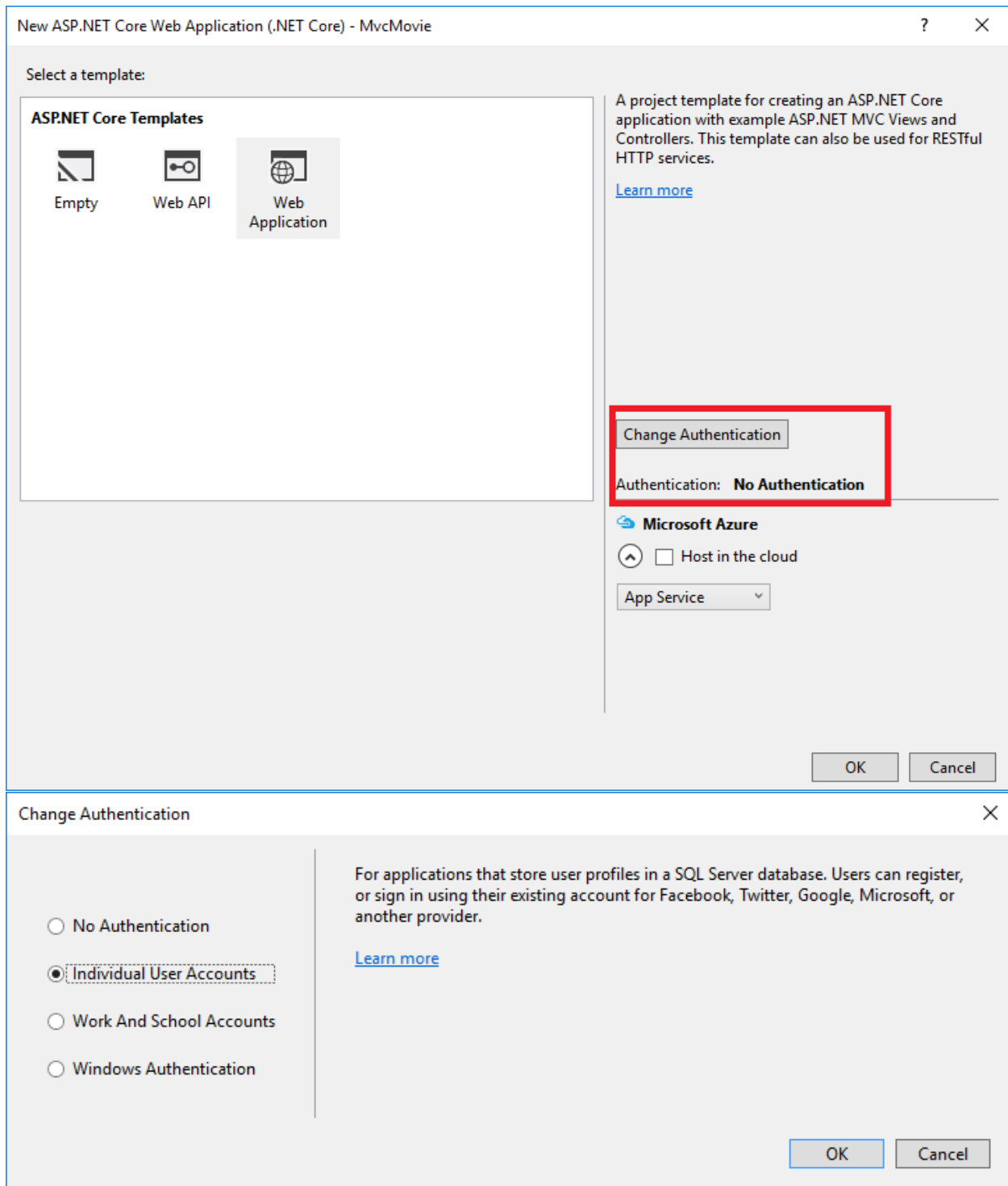
Follow the instructions in *Change the title and menu link in the layout file* so you can tap the **MvcMovie** link to invoke the Movie controller. We'll scaffold the movies controller in this tutorial.

Adding data model classes   In Solution Explorer, right click the *Models* folder > **Add** > **Class**. Name the class **Movie** and add the following properties:

```
using System;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

In addition to the properties you'd expect to model a movie, the `ID` field is required by the DB for the primary key. Build the project. If you don't build the app, you'll get an error in the next section. We've finally added a **M**odel to our **MVC** app.

Scaffolding a controller   In **Solution Explorer**, right-click the *Controllers* folder **> Add > Controller**.
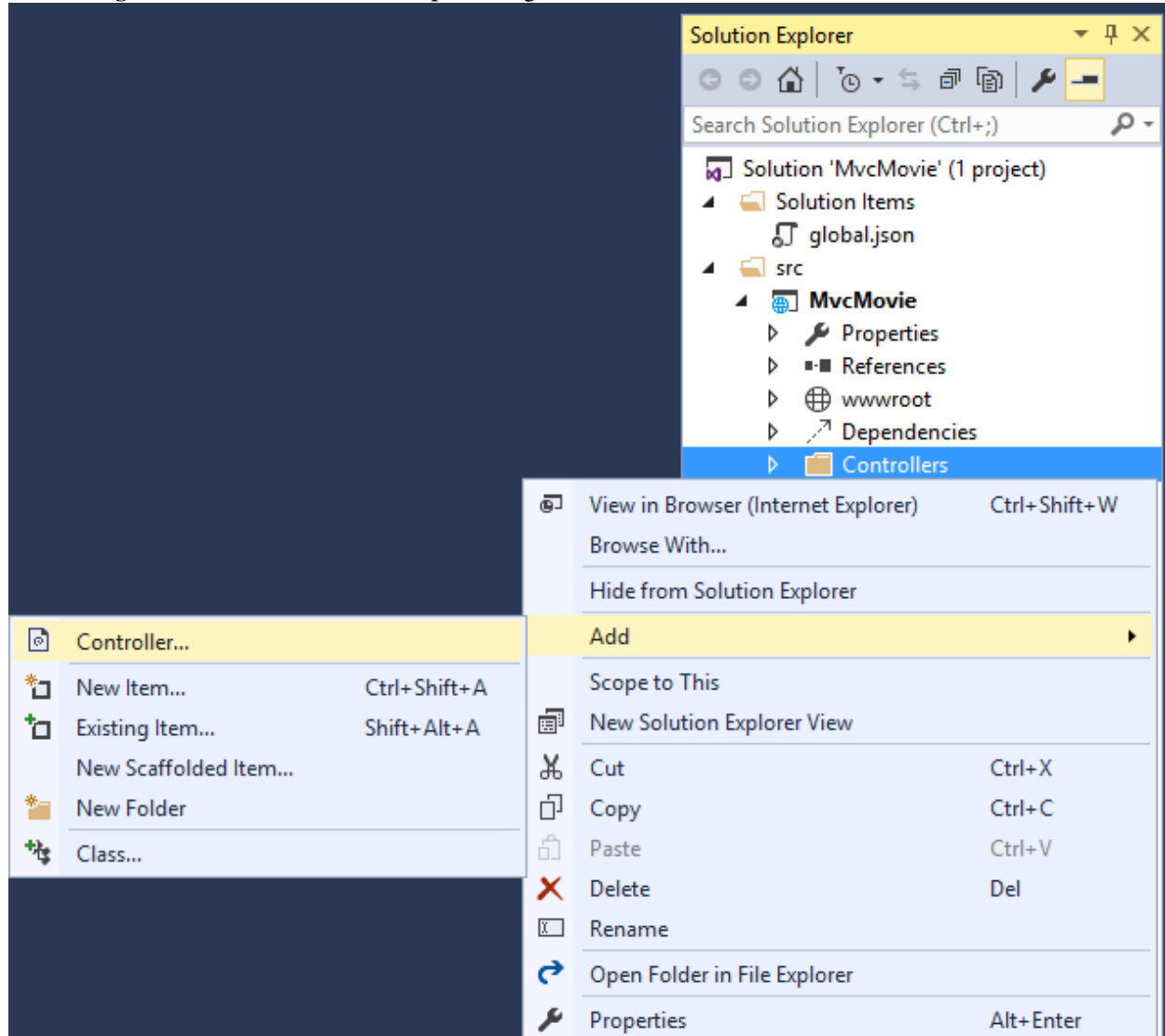
In the **Add Scaffold** dialog, tap **MVC Controller with views, using Entity Framework > Add**.
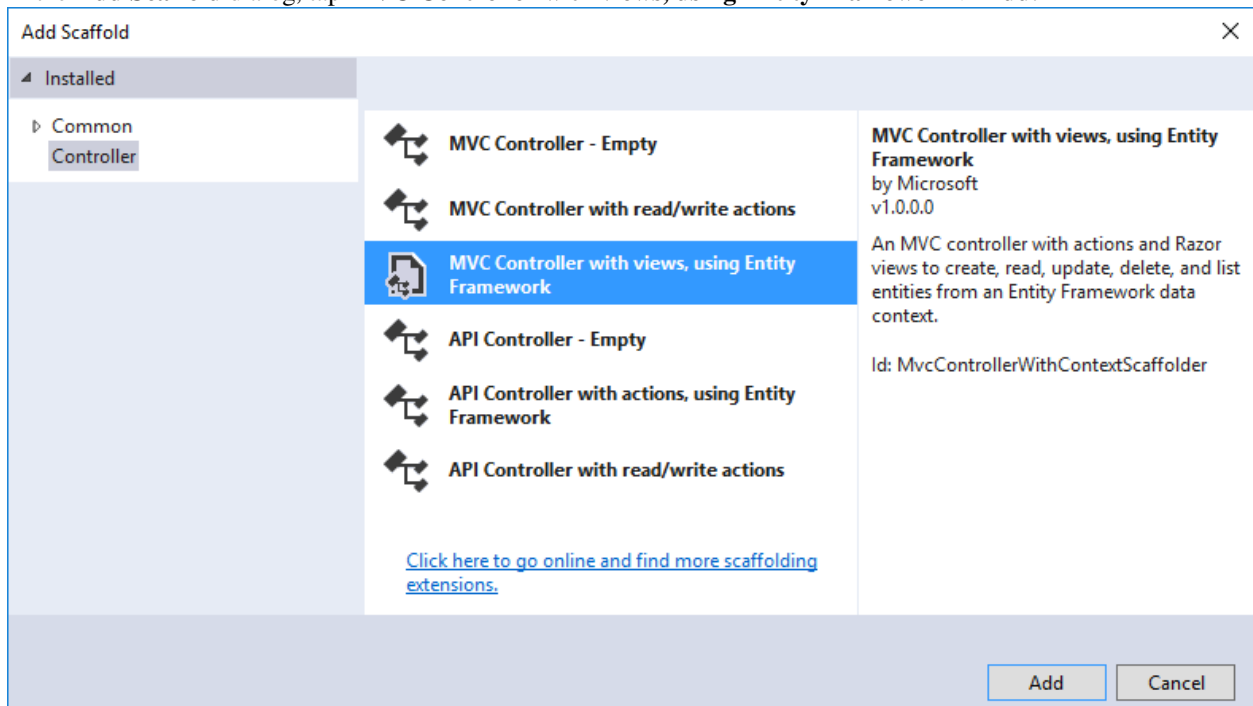
Complete the **Add Controller** dialog

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

The Visual Studio scaffolding engine creates the following:

- A movies controller (*Controllers/MoviesController.cs*)

- Create, Delete, Details, Edit and Index Razor view files (*Views/Movies*)

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

If you run the app and click on the **Mvc Movie** link, you'll get the following errors:
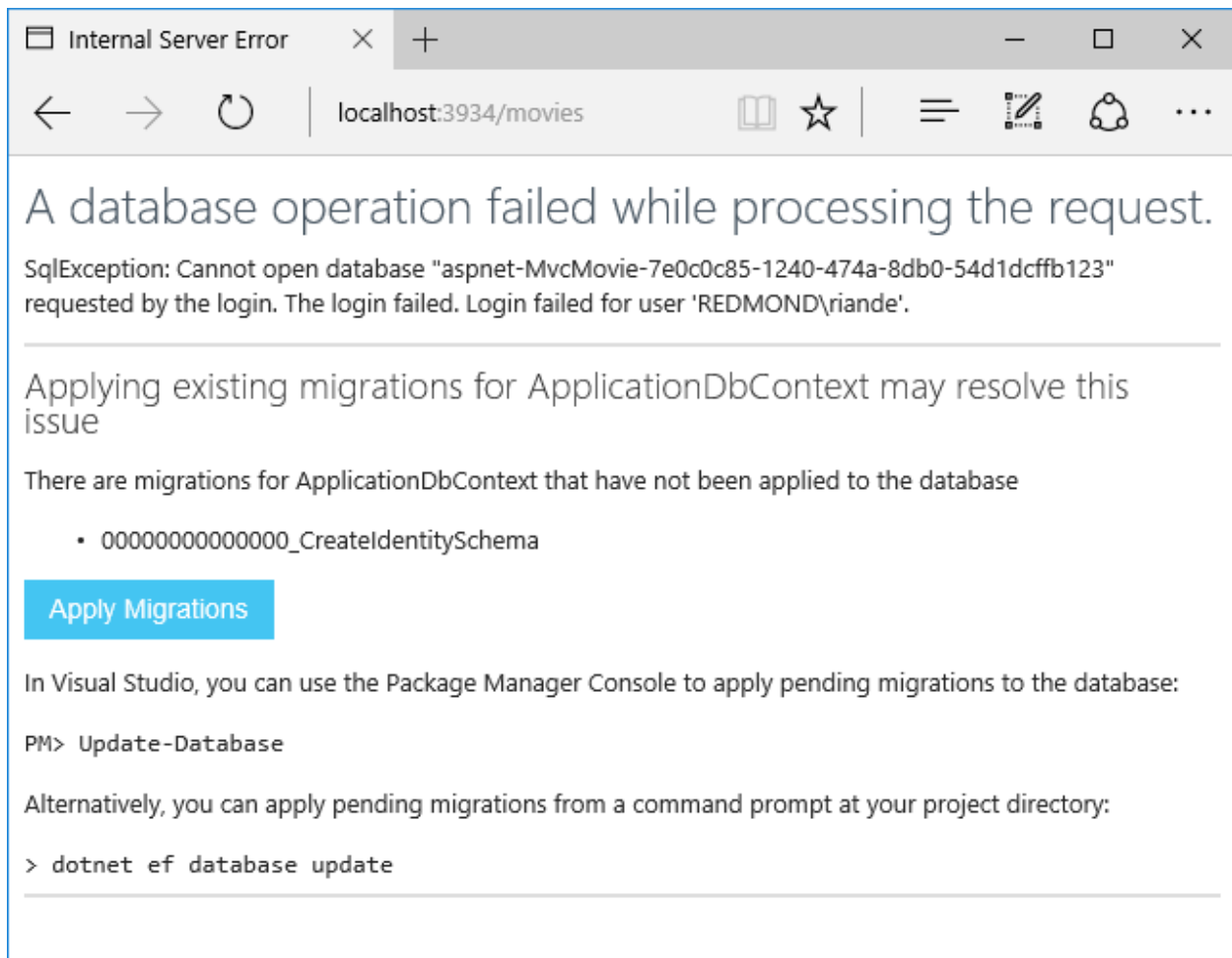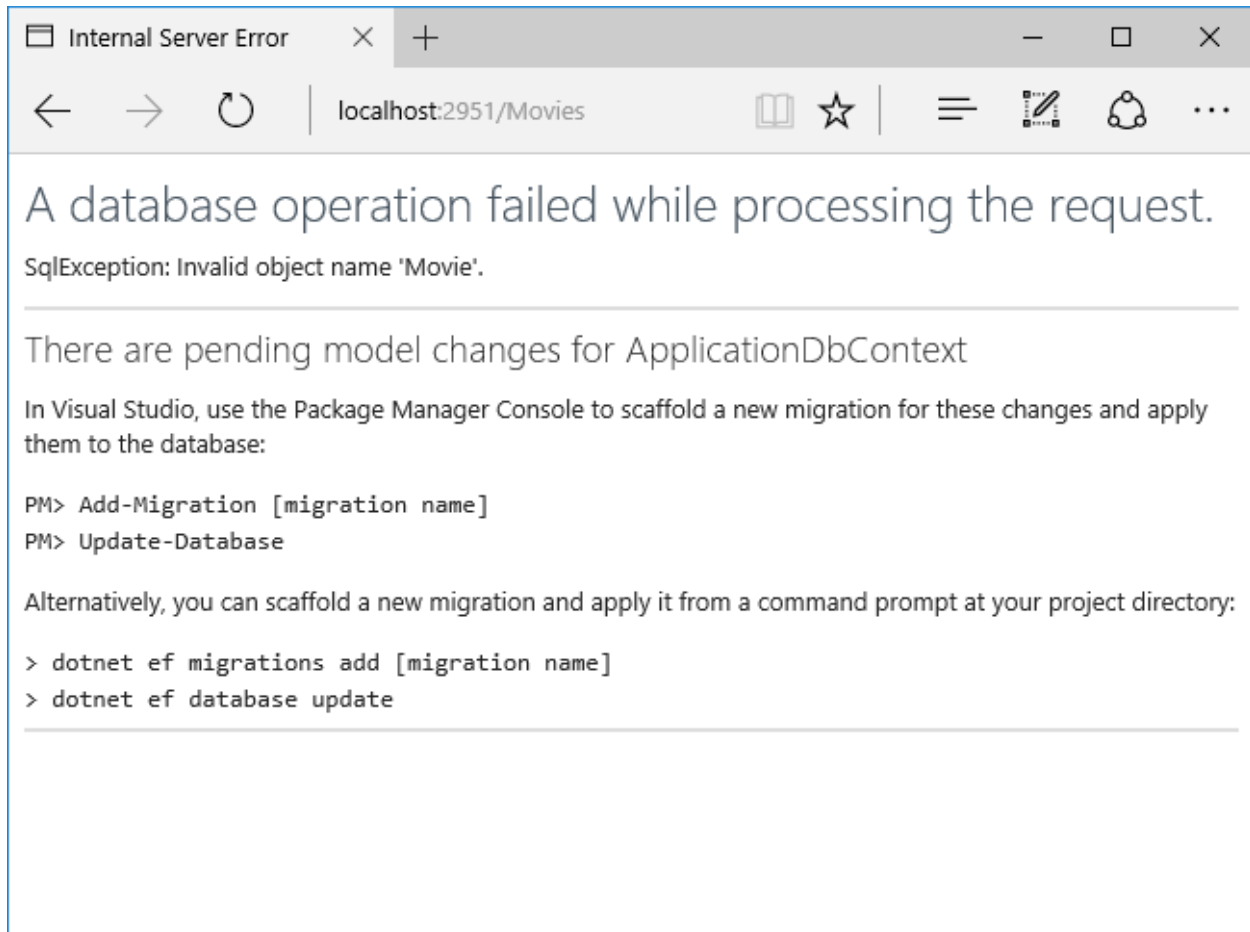
We'll follow those instructions to get the database ready for our Movie app.

Update the database	**:** You must stop IIS Express before you update the database.

To Stop IIS Express:

- Right click the IIS Express system tray icon in the notification area
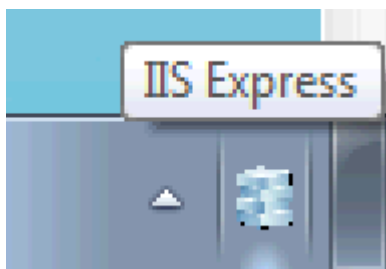
- Tap **Exit** or **Stop Site**
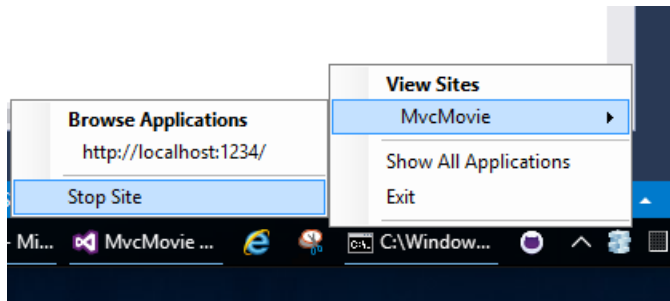
- Alternatively, you can exit and restart Visual Studio

- Open a command prompt in the project directory (MvcMovie/src/MvcMovie). Follow these instructions for a quick way to open a folder in the project directory.

    - Open a file in the root of the project (for this example, use *Startup.cs*.)

    - Right click on *Startup.cs* **> Open Containing Folder**.
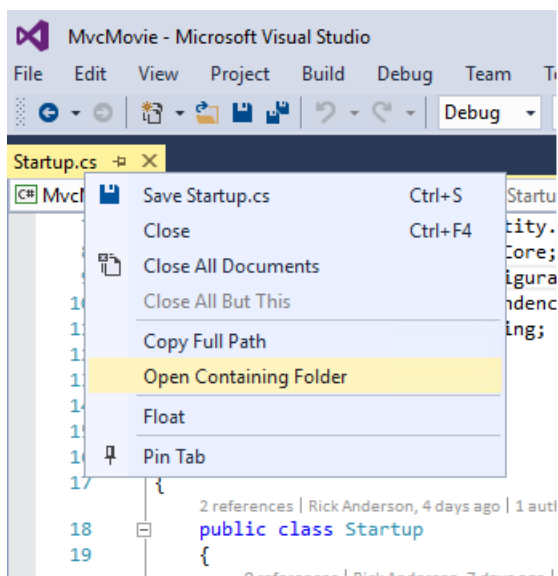
- Shift + right click a folder **> Open command window here**
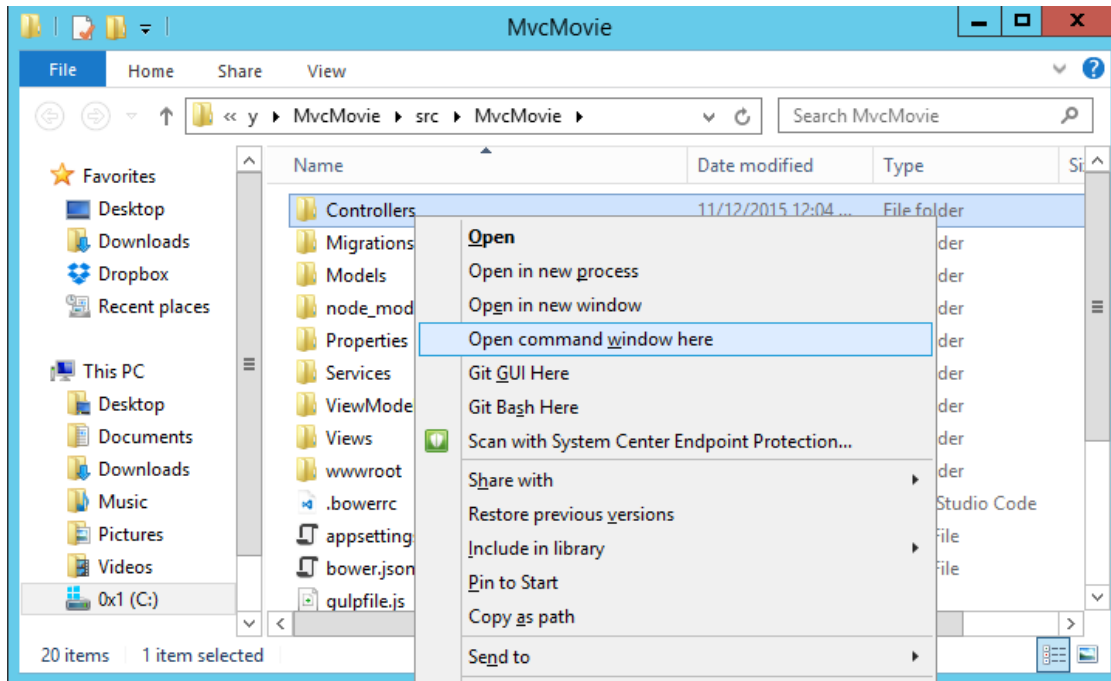
- Run `cd ..` to move back up to the project directory

- Run the following commands in the command prompt:

```
dotnet ef migrations add Initial
dotnet ef database update
```

: If IIS-Express is running, you'll get the error *CS2012: Cannot open 'Mvc-Movie/bin/Debug/netcoreapp1.0/MvcMovie.dll' for writing – 'The process cannot access the file 'Mvc-Movie/bin/Debug/netcoreapp1.0/MvcMovie.dll' because it is being used by another process.'*

dotnet ef commands

- `dotnet` (.NET Core) is a cross-platform implementation of .NET. You can read about it here

- `dotnet ef migrations add Initial` Runs the Entity Framework .NET Core CLI migrations command and creates the initial migration. The parameter "Initial" is arbitrary, but customary for the first (*initial*) database migration. This operation creates the *Data/Migrations/<date-time>_Initial.cs* file containing the migration commands to add (or drop) the *Movie* table to the database

- `dotnet ef database update` Updates the database with the migration we just created

Test the app

: If your browser is unable to connect to the movie app you might need to wait for IIS Express to load the app. It can sometimes take up to 30 seconds to build the app and have it ready to respond to requests.

- Run the app and tap the **Mvc Movie** link
- Tap the **Create New** link and create a movie

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

: In some locals you'll need to specify the date format. See the highlighted code below.

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

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

Tapping **Create** causes the form to be posted to the server, where the movie information is saved in a database. You are then redirected to the */Movies* URL, where you can see the newly created movie in the listing.
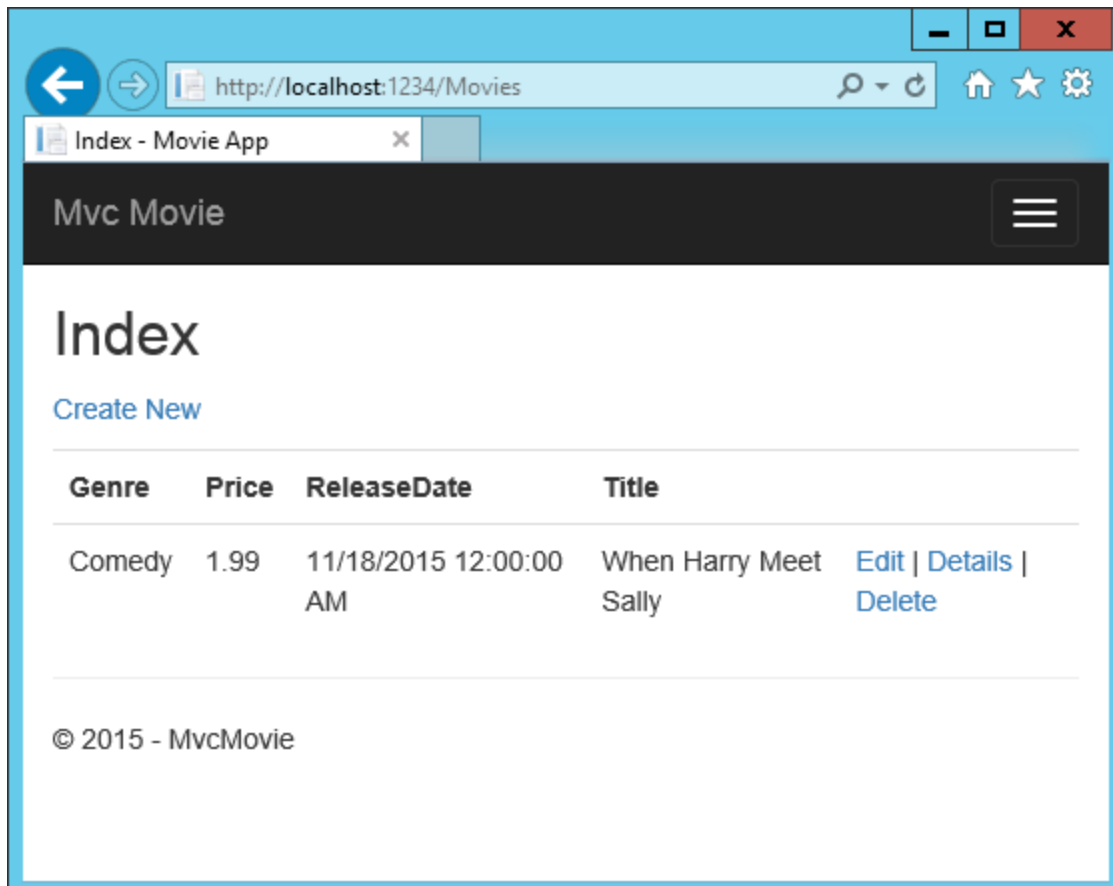
Create a couple more movie entries. Try the **Edit**, **Details**, and **Delete** links, which are all functional.

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

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

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

    // GET: Movies
    public async Task<IActionResult> Index()
    {
        return View(await _context.Movie.ToListAsync());
    }
```

The constructor uses *Dependency Injection* to inject the database context into the controller. The database context is used in each of the CRUD methods in the controller.

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

Strongly typed models and the @model keyword

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

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

Examine the generated `Details` method in the *Controllers/MoviesController.cs* file:

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

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

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

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

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

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

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

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

```
@model MvcMovie.Models.Movie

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

<h2>Details</h2>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Price)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Title)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.ID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

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

---

top of the *Details.cshtml* file:

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

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

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

```csharp
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

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

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

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

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

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

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
```

```
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

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

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

Additional resources

- *Tag Helpers*

- *Globalization and localization*

Working with SQL Server LocalDB

By Rick Anderson

The `ApplicationDbContext` class handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the *Dependency Injection* container in the `ConfigureServices` method in the *Startup.cs* file:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
```

The ASP.NET Core *Configuration* system reads the `ConnectionString`. For local development, it gets the connection string from the *appsettings.json* file:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-MvcMovie-4ae3798a;Trusted_Co
  },
  "Logging": {
    "IncludeScopes": false,
```

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

SQL Server Express LocalDB

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

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).
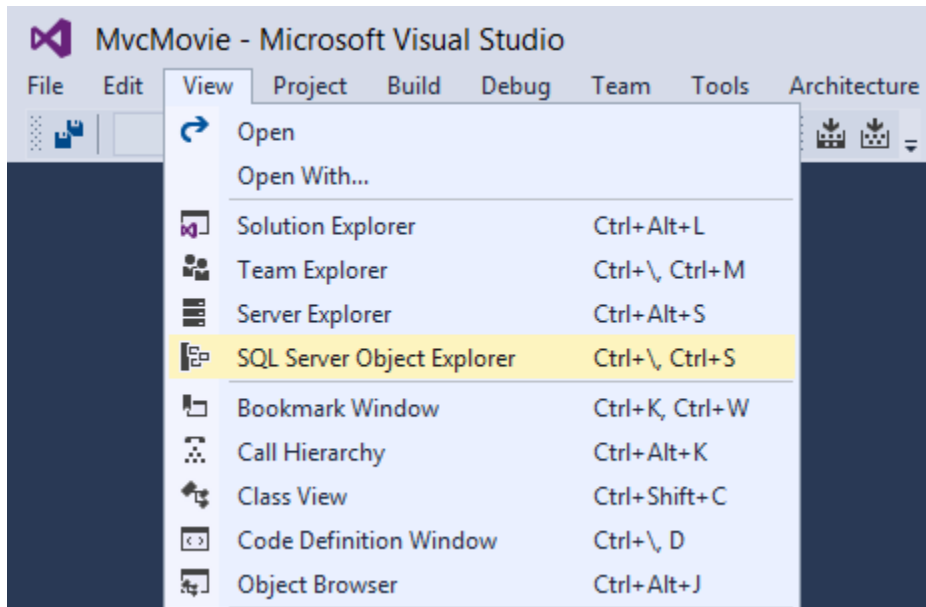
- Right click on the `Movie` table **> View Designer**
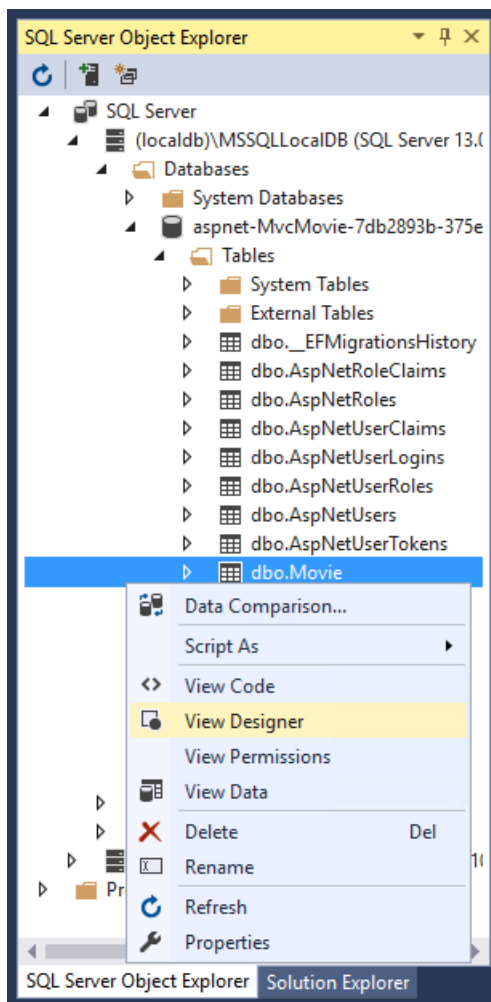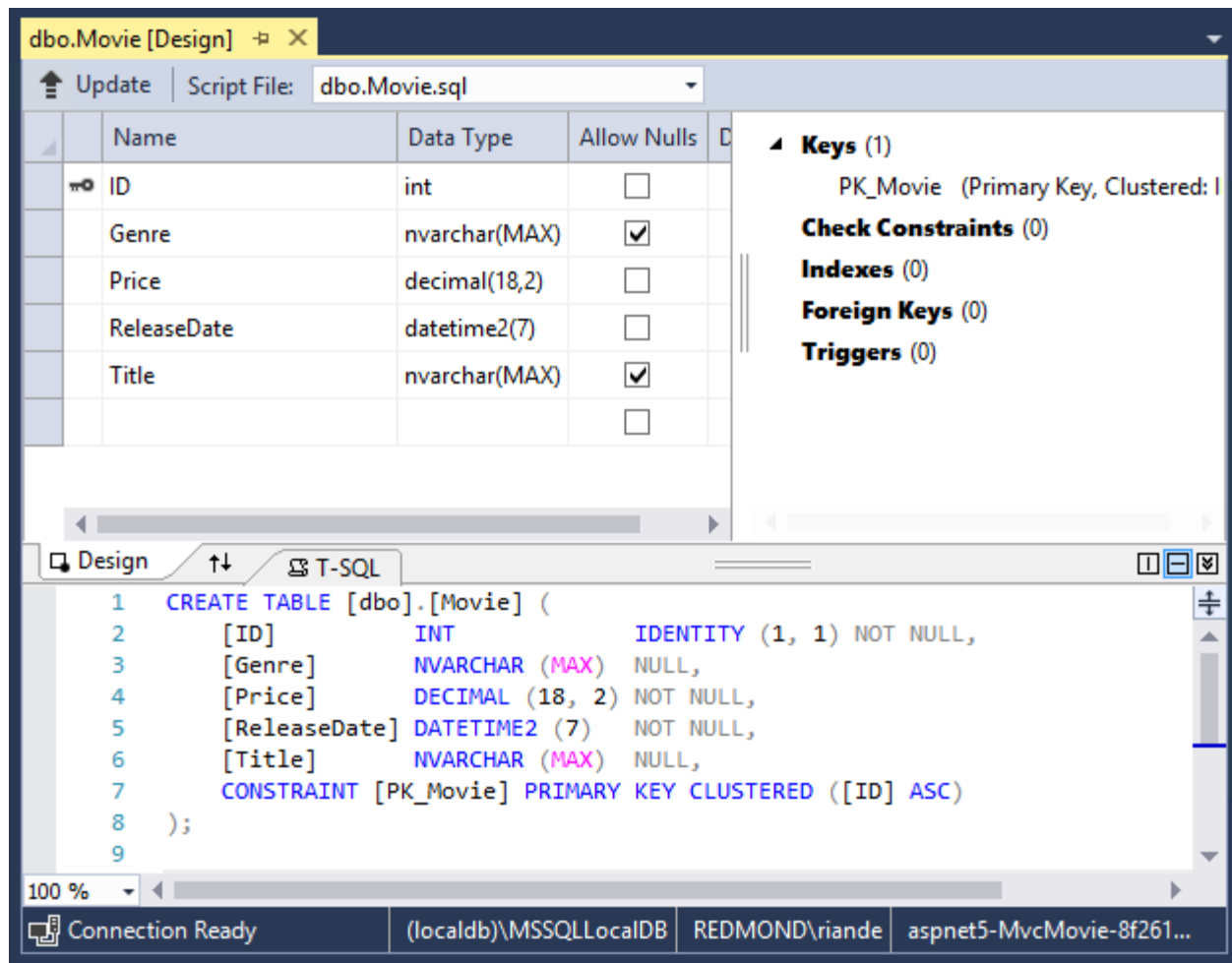
Note the key icon next to `ID`. By default, EF will make a property named `ID` the primary key.

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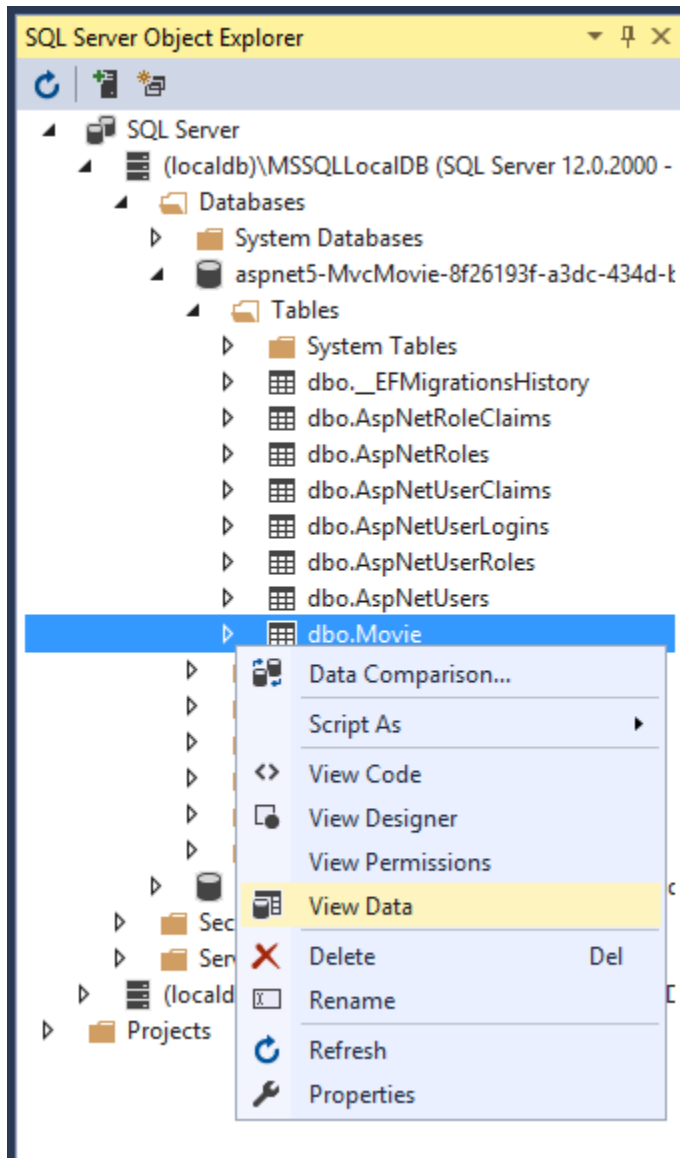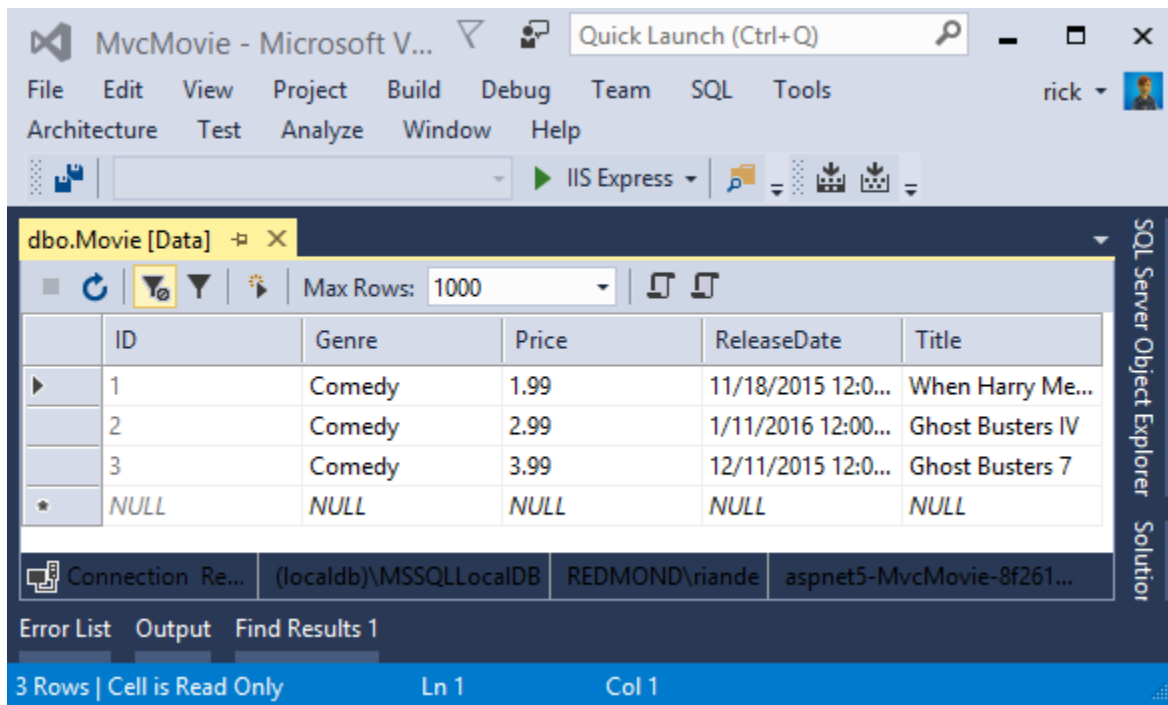
Seed the database

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

```csharp
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new ApplicationDbContext(
                serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

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

```
            },

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

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

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

Notice if there are any movies in the DB, the seed initializer returns.

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

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

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

    SeedData.Initialize(app.ApplicationServices);
}
```

Test the app

- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX.

- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:

    - Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**
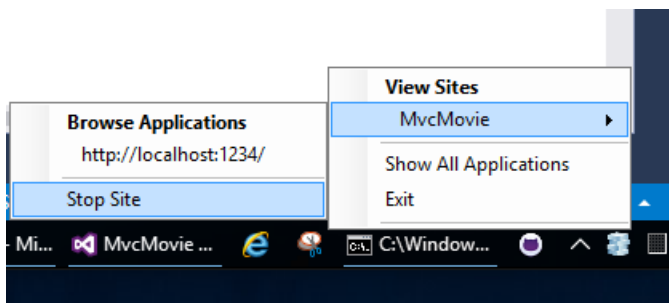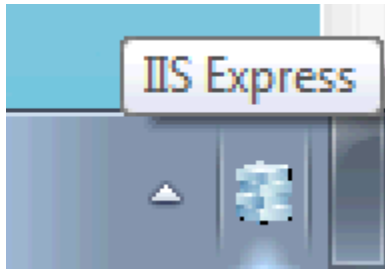
- If you were running VS in non-debug mode, press F5 to run in debug mode
- If you were running VS in debug mode, stop the debugger and press ^F5

: If the database doesn't initialize, put a break point on the line `if (context.Movie.Any())` and start debugging.
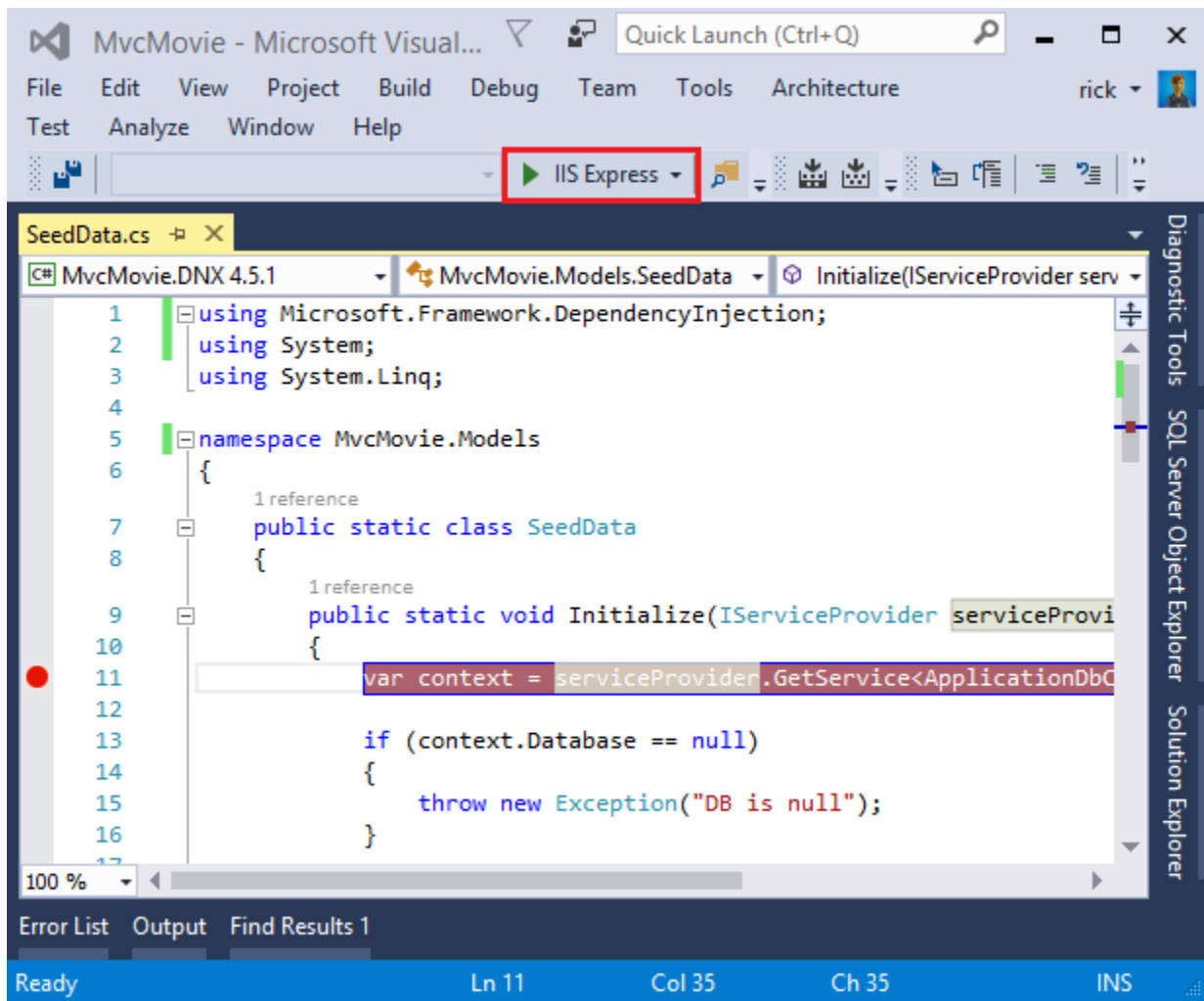
The app shows the seeded data.
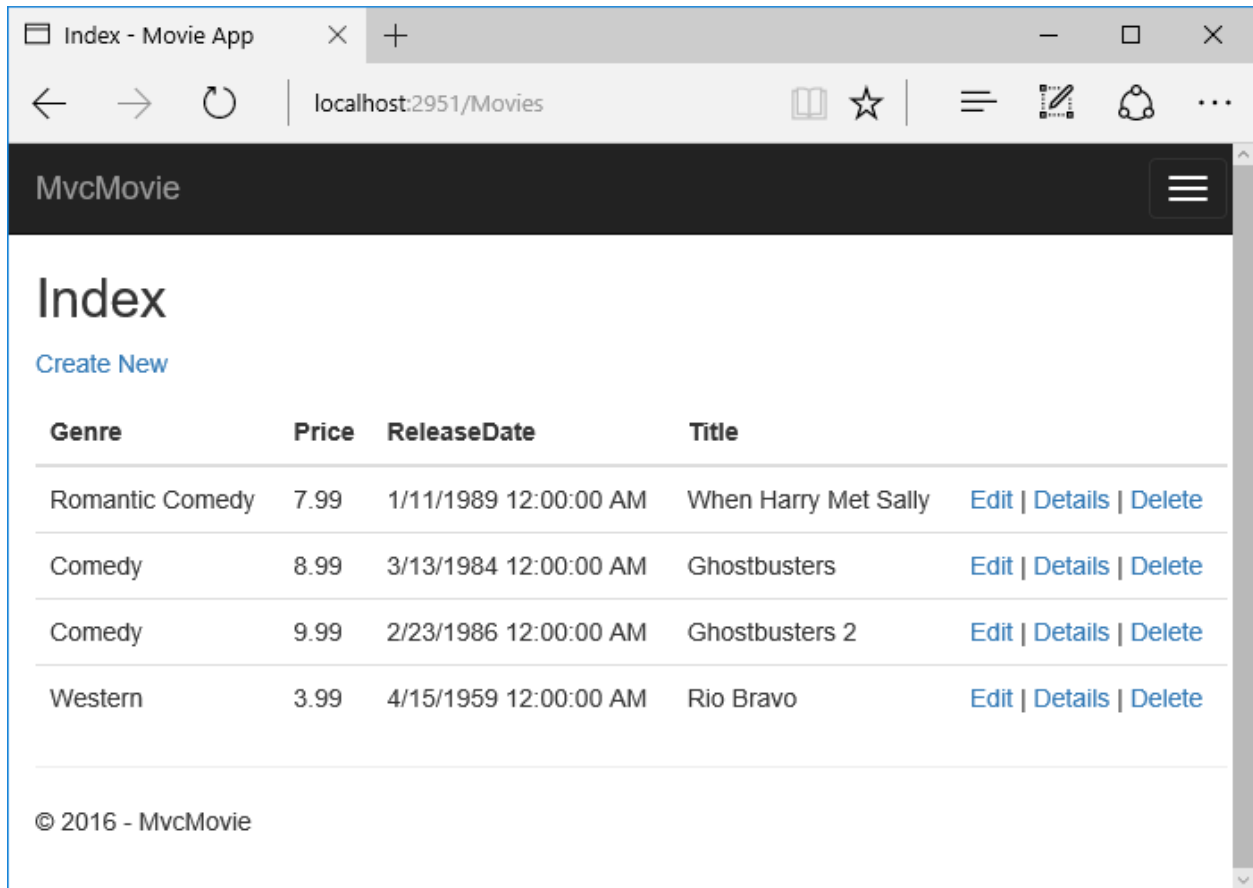
Controller methods and views

By Rick Anderson

We have a good start to the movie app, but the presentation is not ideal. We don't want to see the time (12:00:00 AM in the image below) and **ReleaseDate** should be two words.
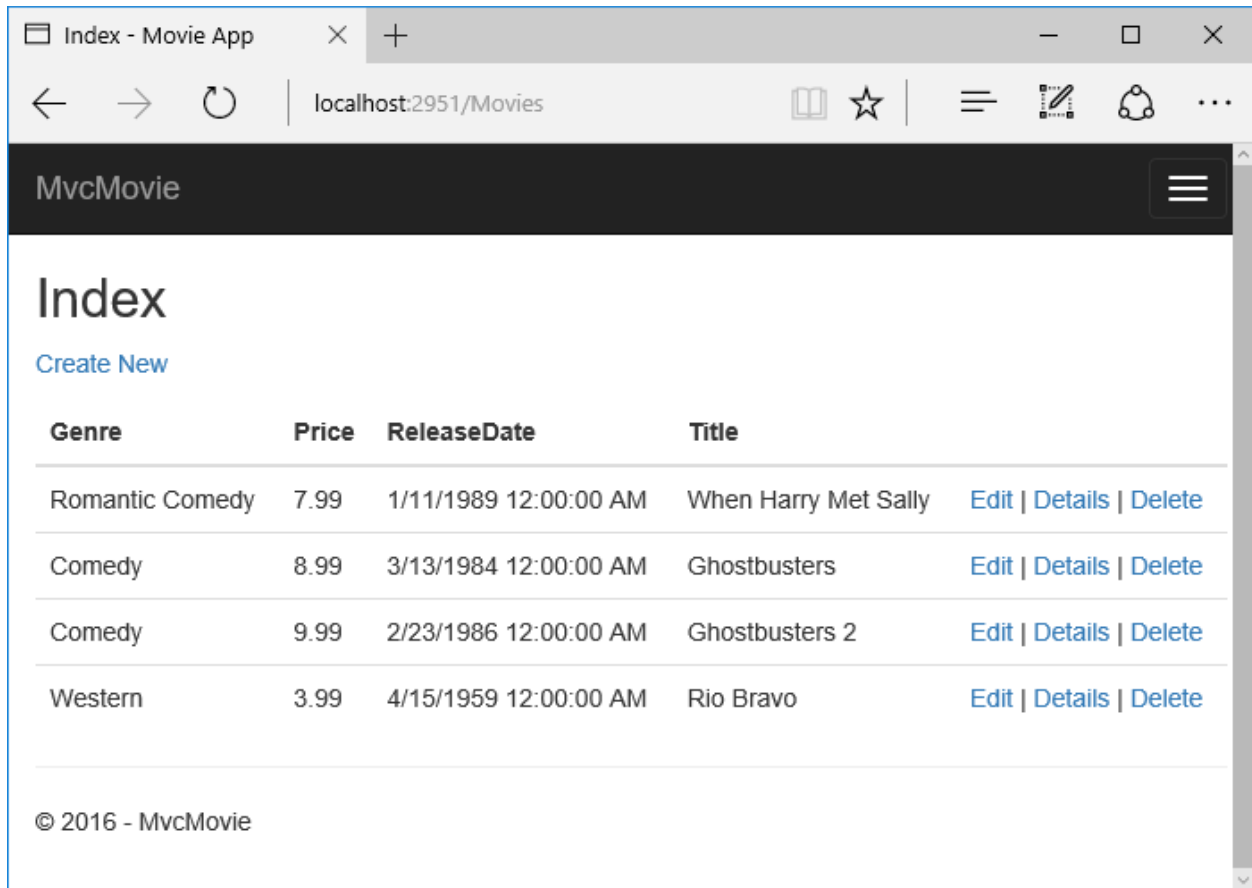
Open the *Models/Movie.cs* file and add the highlighted lines shown below:

```csharp
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

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

- Right click on a red squiggly line **> Quick Actions**.
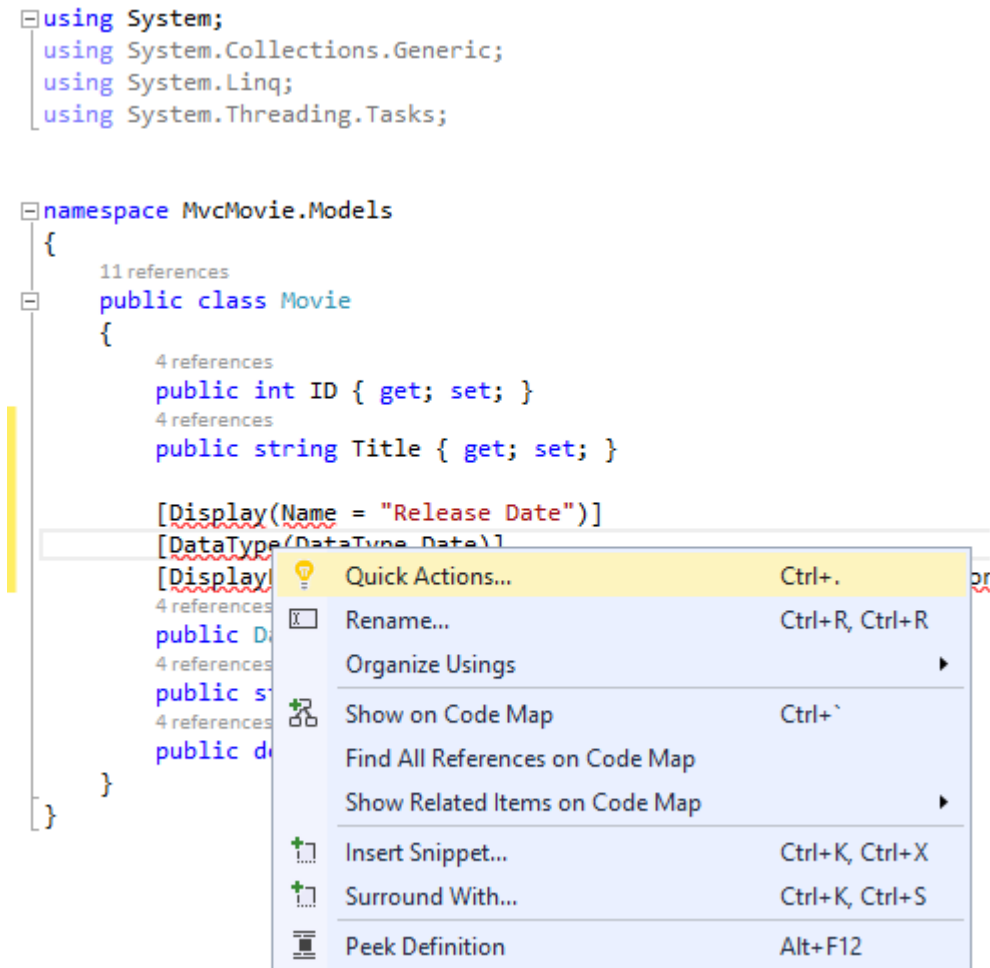
- Tap `using System.ComponentModel.DataAnnotations;`
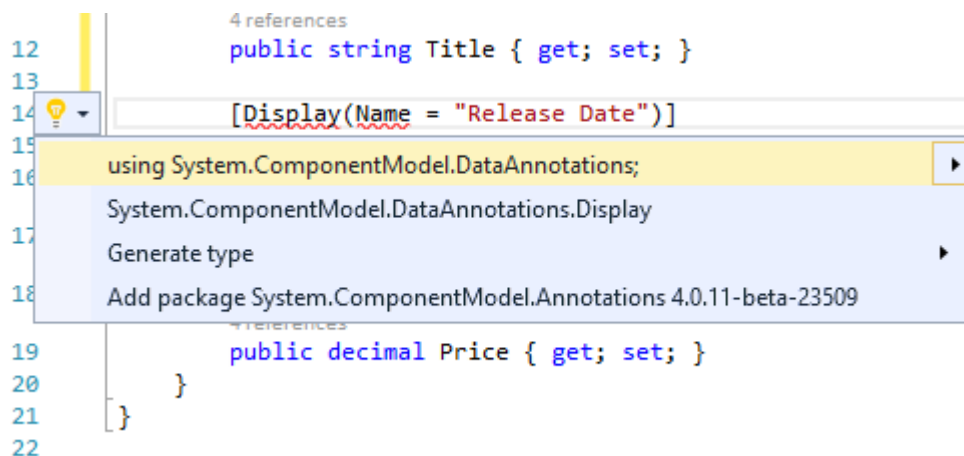
Visual studio adds `using System.ComponentModel.DataAnnotations;`.

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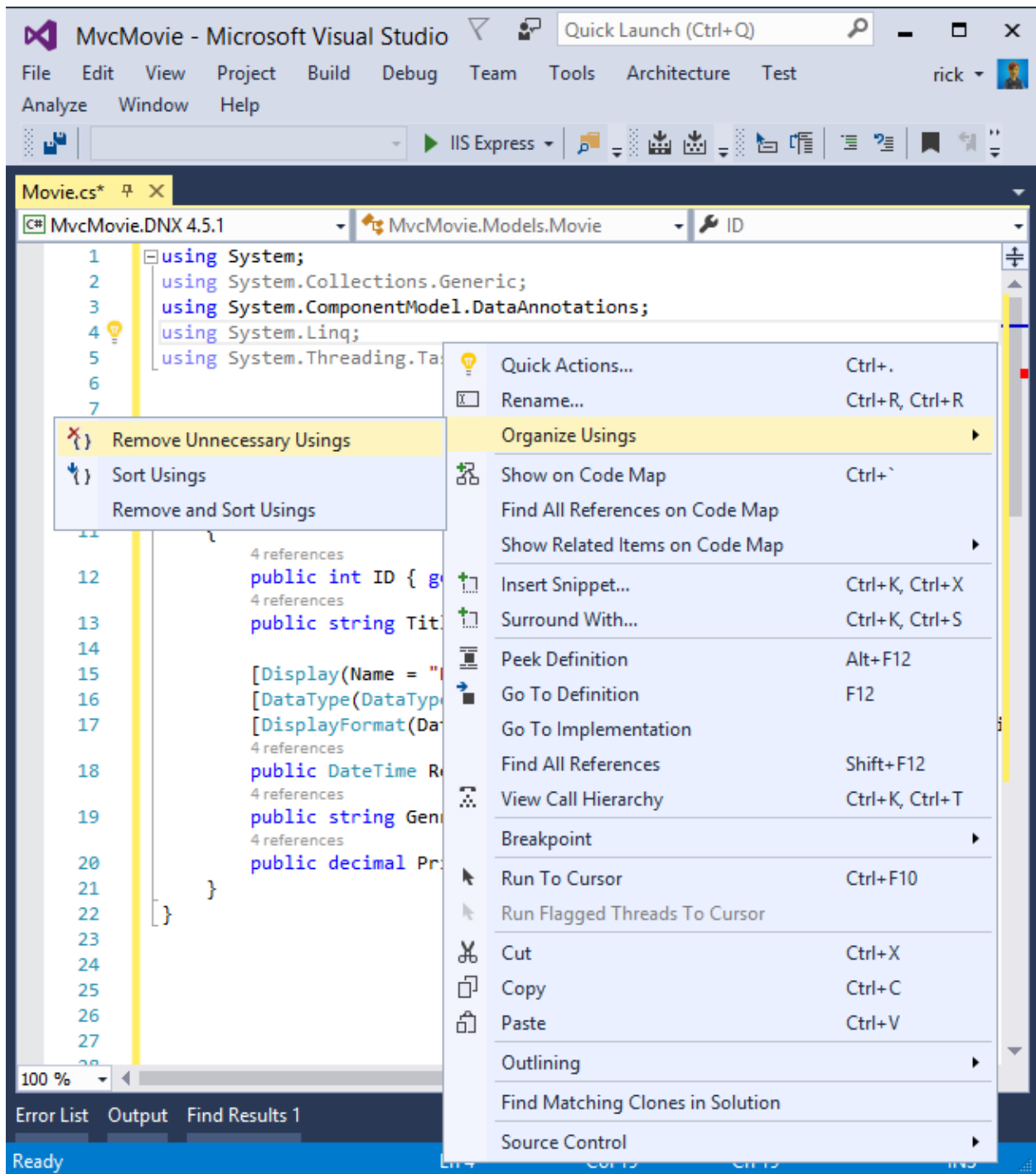
The updated code:

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

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
```

```
        public string Title { get; set; }

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

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

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

The **Edit**, **Details**, and **Delete** links are generated by the MVC Core Anchor Tag Helper in the *Views/Movies/Index.cshtml* file.

```
<td>
    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
</td>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code

above, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the **F12** tools to examine the generated markup. The **F12** tools are shown below.
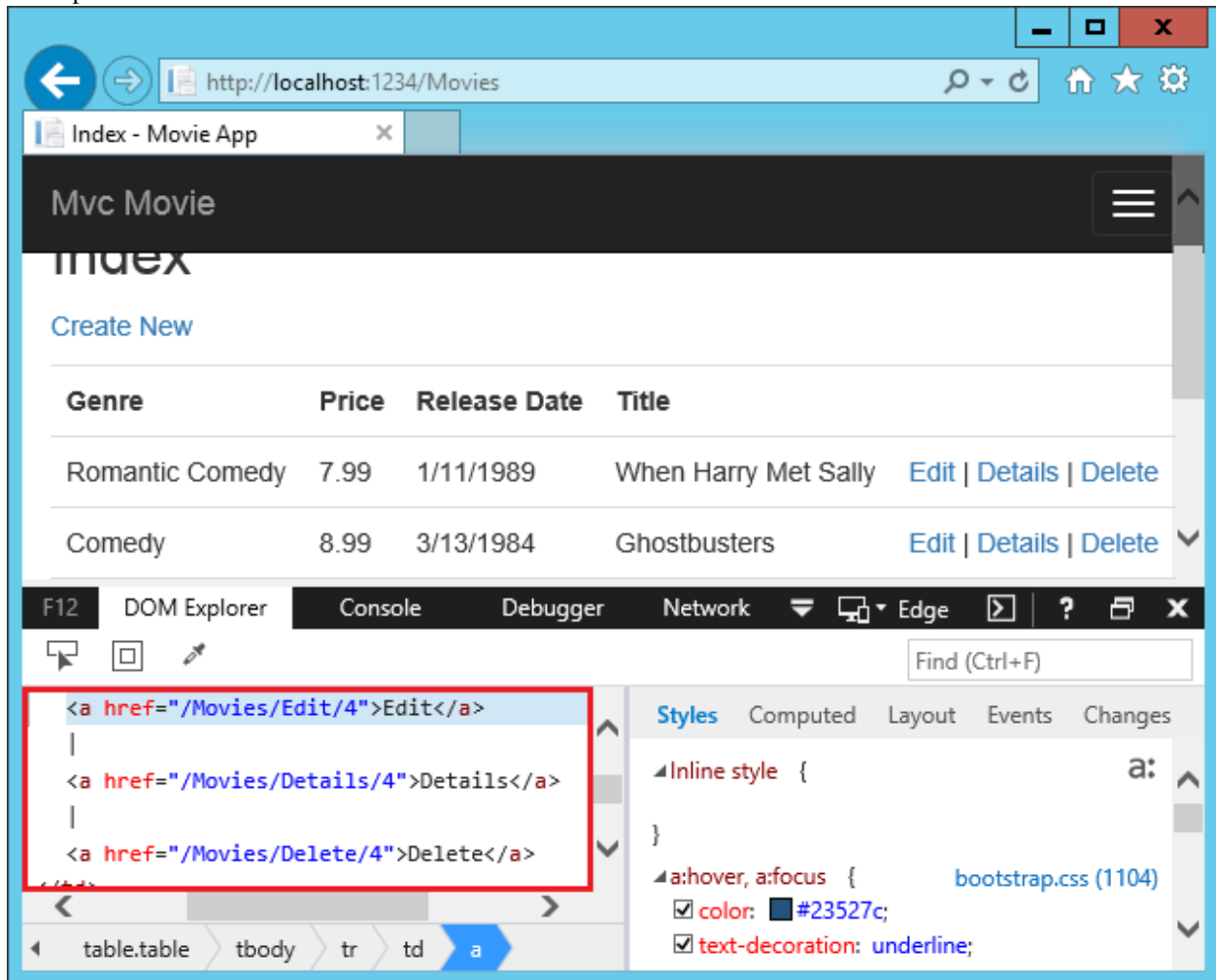
Recall the format for routing set in the *Startup.cs* file.

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

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

Tag Helpers are one of the most popular new features in ASP.NET Core. See *Additional resources* for more information.

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

```csharp
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

```csharp
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Genre,Price,ReleaseDate,Title")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }
```

```csharp
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The [Bind] attribute is one way to protect against over-posting. You should only include properties in the [Bind] attribute that you want to change. See Protect your controller from over-posting for more information. ViewModels provide an alternative approach to prevent over-posting.

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

```csharp
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Genre,Price,ReleaseDate,Title")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
```

```
}
```

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

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

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

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

The `HttpGet Edit` method takes the movie `ID` parameter, looks up the movie using the Entity Framework `SingleOrDefaultAsync` method, and returns the selected movie to the Edit view. If a movie cannot be found, `NotFound` (HTTP 404) is returned.

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

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

```
@model MvcMovie.Models.Movie

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

<h2>Edit</h2>

<form asp-action="Edit">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <input type="hidden" asp-for="ID" />
        <div class="form-group">
            <label asp-for="Genre" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger" />
            </div>
```

```
            </div>
            <div class="form-group">
                <label asp-for="Price" class="col-md-2 control-label"></label>
                <div class="col-md-10">
                    <input asp-for="Price" class="form-control" />
                    <span asp-validation-for="Price" class="text-danger" />
                </div>
            </div>
            <div class="form-group">
                <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
                <div class="col-md-10">
                    <input asp-for="ReleaseDate" class="form-control" />
                    <span asp-validation-for="ReleaseDate" class="text-danger" />
                </div>
            </div>
            <div class="form-group">
                <label asp-for="Title" class="col-md-2 control-label"></label>
                <div class="col-md-10">
                    <input asp-for="Title" class="form-control" />
                    <span asp-validation-for="Title" class="text-danger" />
                </div>
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <input type="submit" value="Save" class="btn btn-default" />
                </div>
            </div>
        </div>
    </div>
</form>

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

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

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

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

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

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field is required." id="ID" na
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
                <input class="form-control" type="text" id="Genre" name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg-
```

```
                </div>
            </div>
            <div class="form-group">
                <label class="control-label col-md-2" for="Price" />
                <div class="col-md-10">
                    <input class="form-control" type="text" data-val="true" data-val-number="The field P
                    <span class="text-danger field-validation-valid" data-valmsg-for="Price" data-valmsg-
                </div>
            </div>
            <!-- Markup removed for brevity -->
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <input type="submit" value="Save" class="btn btn-default" />
                </div>
            </div>
        </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden" value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1I
</form>
```

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

### Processing the POST Request

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

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Genre,Price,ReleaseDate,Title")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
```

```
}
```

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

The *model binding* system takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` method verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid it's saved. The updated (edited) movie data is saved to the database by calling the `SaveChangesAsync` method of database context. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form is not posted. If JavaScript is disabled, you won't have client side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine *Model Validation* validation in more detail. The *Validation Tag Helper* in the *Views/Book/Edit.cshtml* view template takes care of displaying appropriate error messages.

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

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

Additional resources

- *Globalization and localization*
- *Introduction to Tag Helpers*
- *Authoring Tag Helpers*
- *Anti-Request Forgery*
- Protect your controller from over-posting
- ViewModels
- *Form Tag Helper*
- *Input Tag Helper*
- *Label Tag Helper*
- *Select Tag Helper*
- *Validation Tag Helper*

Adding Search

By Rick Anderson

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

Update the `Index` action method to enable search:

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

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

    return View(await movies.ToListAsync());
}
```

The first line of the `Index` action method creates a LINQ query to select the movies:

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

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

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

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

The `s => s.Title.Contains()` code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` used in the code above. LINQ queries are not executed when they are defined or when they are modified by calling a method such as `Where`, `Contains` or `OrderBy`. Instead, query execution is deferred, which means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToListAsync` method is called. For more information about deferred query execution, see Query Execution.

: The Contains method is run on the database, not the c# code above. On the database, Contains maps to SQL LIKE, which is case insensitive.

Navigate to `/Movies/Index`. Append a query string such as `?searchString=ghost` to the URL. The filtered movies are displayed.
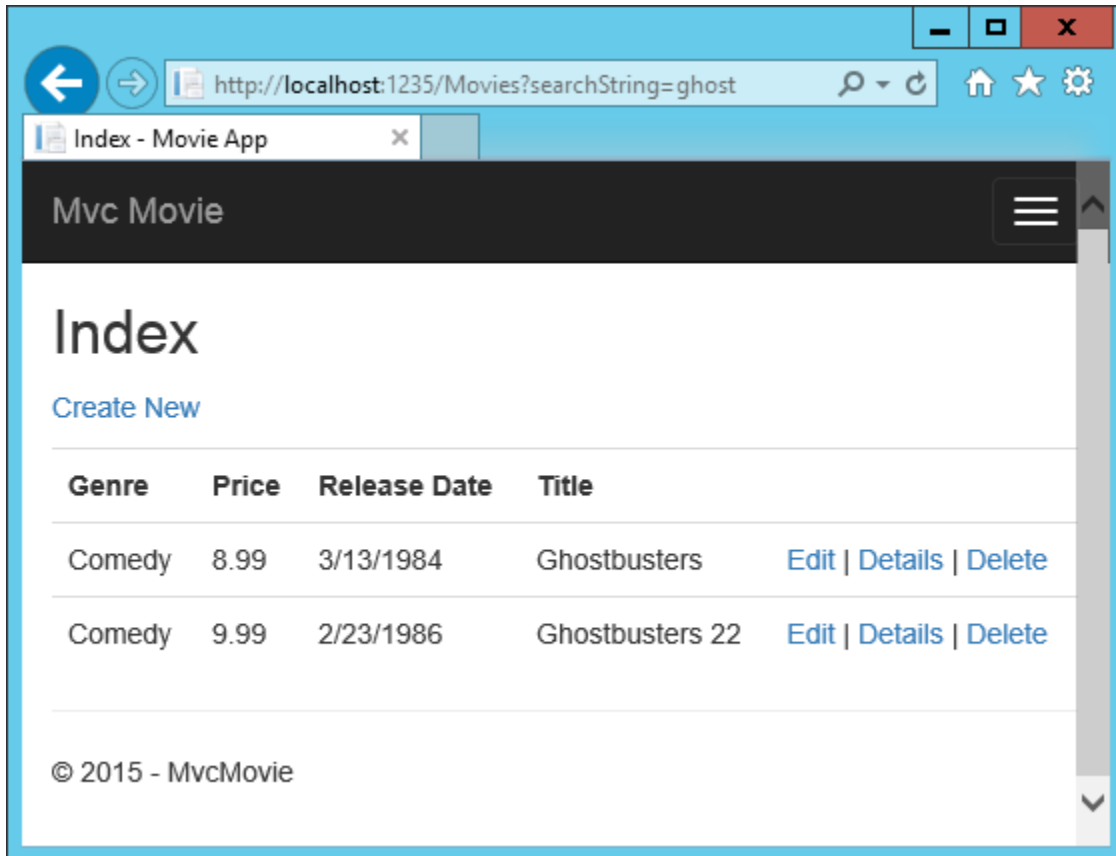
If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in *Startup.cs*.
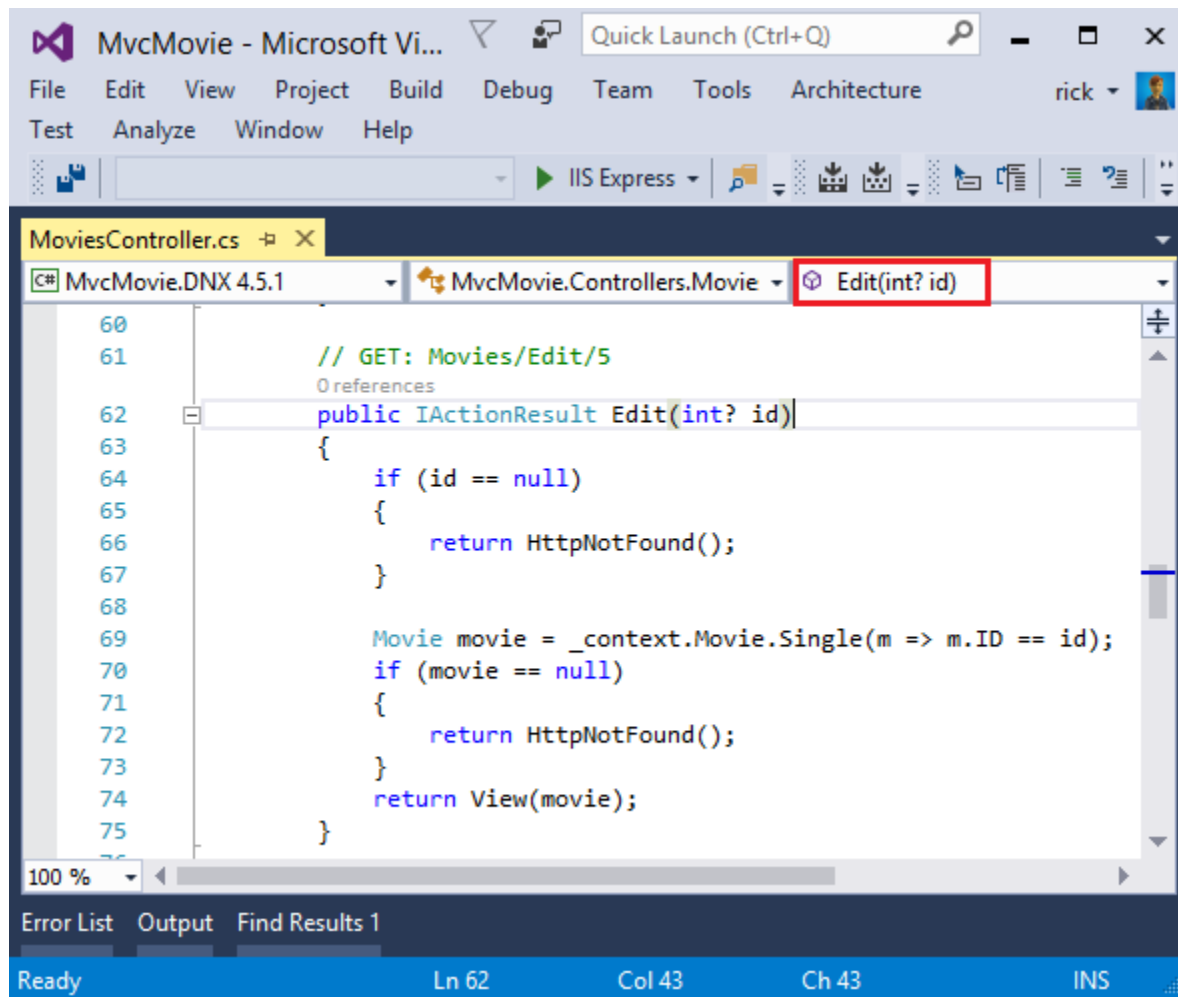
```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

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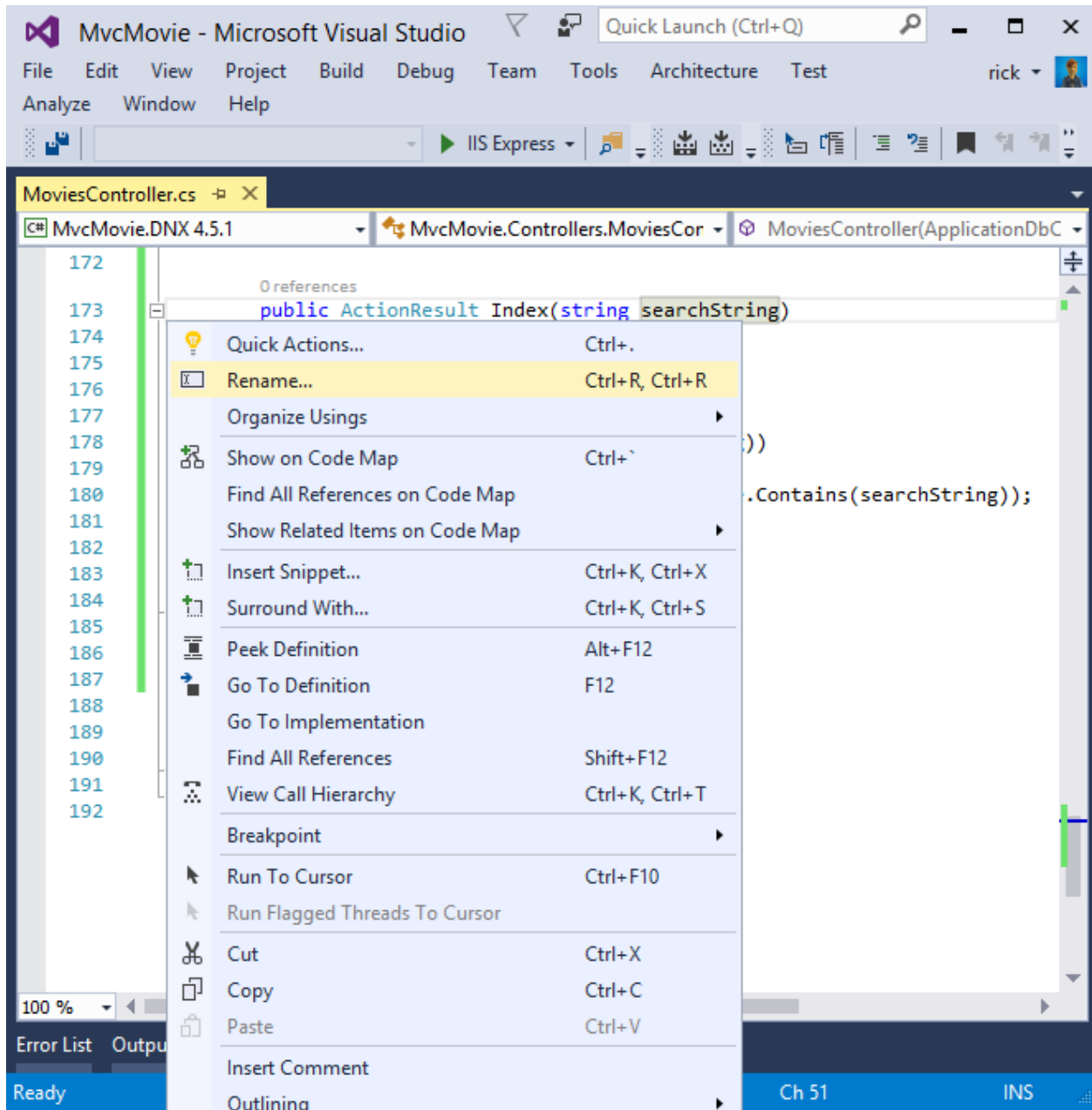
The rename targets are highlighted.

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

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

    return View(movies);
}
```

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

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

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

    return View(movies);
}
```

The previous `Index` method:

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

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

    return View(await movies.ToListAsync());
}
```

The updated `Index` method:

```csharp
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

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

    return View(await movies.ToListAsync());
}
```

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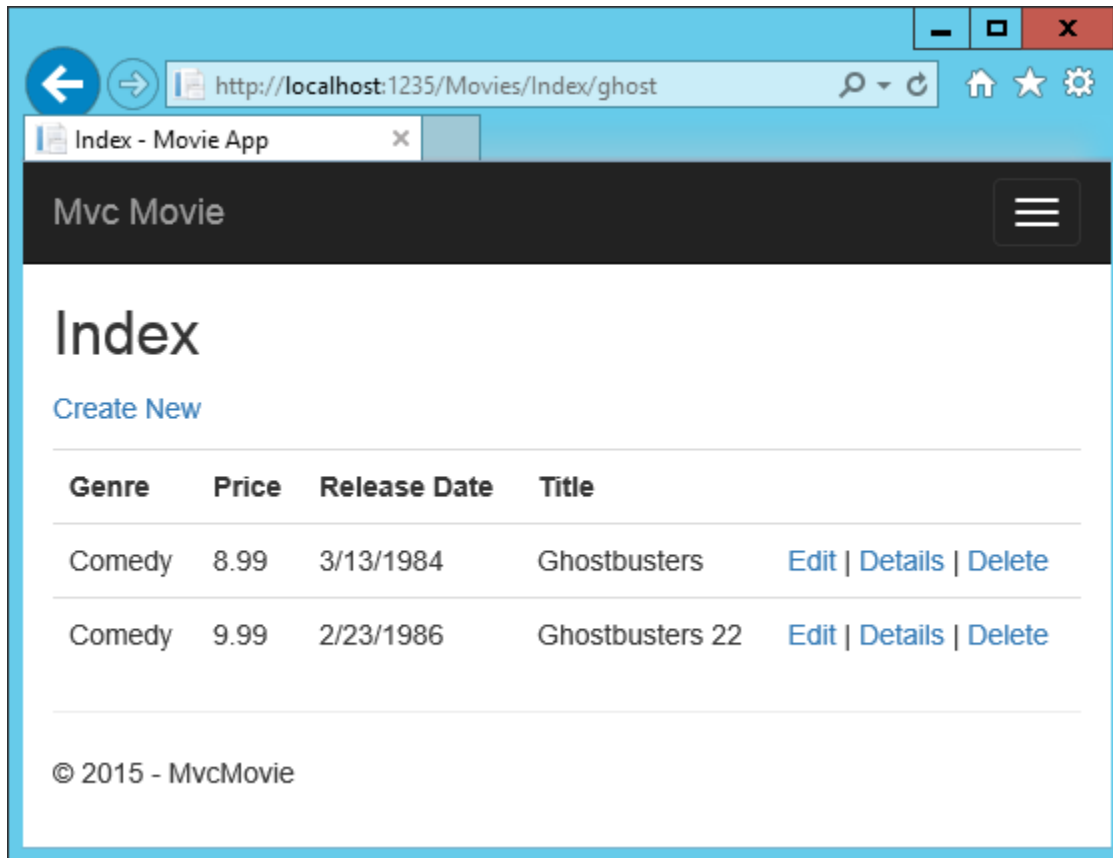
However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound `ID` parameter, change it back so that it takes a parameter named `searchString`:

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

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

    return View(await movies.ToListAsync());
}
```

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

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

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
```

```
</p>

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

<table class="table">
```

The HTML `<form>` tag uses the *Form Tag Helper*, so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.

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

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

```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

The `notUsed` parameter is used to create an overload for the `Index` method. We'll talk about that later in the tutorial.

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

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

Tap the **http://localhost:xxx/Movies HTTP POST 200** line and then tap **Body > Request Body**.

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

token in the controller method.

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

```
<form asp-controller="Movies" asp-action="Index" m>
    <p>
        Title: <input type="text" name="SearchStr
        <input type="submit" value="Filter" />
    </p>
</form>
```
```
  autocomplete
  contextmenu
  itemid
  itemprop
  itemref
  itemscope
  itemtype
  method
  name
```

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

Notice the distinctive font in the `<form>` tag. That distinctive font indicates the tag is supported by *Tag Helpers*.

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

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

The following markup shows the change to the `form` tag:

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

Adding Search by Genre

Add the following `MovieGenreViewModel` class to the *Models* folder:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie> movies;
        public SelectList genres;
        public string movieGenre { get; set; }
    }
}
```

The move-genre view model will contain:

- a list of movies

- a SelectList containing the list of genres. This will allow the user to select a genre from the list.

- `movieGenre`, which contains the selected genre

Replace the `Index` method with the following code:

```
public async Task<IActionResult> Index(string movieGenre, string searchString)
{
```

```
    // Use LINQ to get list of genre's.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

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

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

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

    var movieGenreVM = new MovieGenreViewModel();
    movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    movieGenreVM.movies = await movies.ToListAsync();

    return View(movieGenreVM);
}
```

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

```
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

```
movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync())
```

Adding search by genre to the Index view

```
@model MovieGenreViewModel

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

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        <select asp-for="movieGenre" asp-items="Model.genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString">
```

```
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.movies[0].Genre)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.movies[0].Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.movies[0].ReleaseDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.movies[0].Title)
        </th>
        <th></th>
    </tr>
    <tbody>
        @foreach (var item in Model.movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

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

Adding a New Field

By Rick Anderson

In this section you'll use Entity Framework Code First Migrations to add a new field to the model and migrate that change to the database.

When you use EF Code First to automatically create a database, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, EF

throws an exception. This makes it easier to track down issues at development time that you might otherwise only find (by obscure errors) at run time.

Adding a Rating Property to the Movie Model

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

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the app (Ctrl+Shift+B).

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

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

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

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

```
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.movies[0].Genre)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.movies[0].Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.movies[0].ReleaseDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.movies[0].Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.movies[0].Rating)
        </th>
        <th></th>
    </tr>
    <tbody>
        @foreach (var item in Model.movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
```

```html
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Rating)
            </td>
```

Update the */Views/Movies/Create.cshtml* with a `Rating` field. You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with *Tag Helpers*.

```html
        </div>
        <div class="form-group">
            <label asp-for="Title" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger" />
            </div>
        </div>
        <div class="form-group">
            <label asp-for="R" class="col-md-2 control-label"></label>
            <div class="col-          ⊕ GetHashCode          ▲
                <input asp-f        ⊕ GetType              n-control" />
                <span asp-va        🔧 ID                  :lass="text-danger" />
            </div>                   🔧 Price
        </div>                       🔧 Rating
        <div class="form-gro         🔧 ReleaseDate        '>
            <div class="col-         🔧 Title              >" class="btn btn-default" />
            </div>                   ⊕ ToString            ▼
        </div>
    </div>
</form>
```

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

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

There are a few approaches to resolving the error:

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

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

3. Use Code First Migrations to update the database schema.

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

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

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

> **:** You must stop IIS Express before you run the `dotnet ef` commands. See *To Stop IIS Express:*

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

```
dotnet ef migrations add Rating
dotnet ef database update
```

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

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

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

Adding Validation

By Rick Anderson

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

Keeping things DRY

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

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

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

Adding validation rules to the movie model

Open the *Movie.cs* file. DataAnnotations provides a built-in set of validation attributes that you apply declaratively to any class or property. (It also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.)

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

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

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

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
```

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

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]*$")]
    [StringLength(5)]
    public string Rating { get; set; }
}
```

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

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

Validation Error UI in MVC

Run the app and navigate to the Movies controller.

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

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

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

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

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

How Validation Occurs in the Create View and Create Action Method

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

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

// POST: Movies/Create
// To protect from overposting attacks, please enable the specific properties you want to bind to, fo
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("ID,Genre,Price,ReleaseDate,Title,Rating")] Movie movie
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

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

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

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

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

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

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

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

```html
<form asp-action="Create">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <div class="form-group">
            <label asp-for="Genre" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger" />
            </div>
        </div>
        @*Markup removed for brevity.*@
        <div class="form-group">
            <label asp-for="Rating" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Rating" class="form-control" />
                <span asp-validation-for="Rating" class="text-danger" />
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </div>
    </div>
</form>
```

The *Input Tag Helper* consumes the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client side. The *Validation Tag Helper* displays a validation errors. See *Validation* for more information.

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

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

---

honoring the DRY principle.

Using DataType Attributes

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

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public string Genre { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
```

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

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

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

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

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

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

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)

- By default, the browser will render data using the correct format based on your locale

- The `DataType` attribute can enable MVC to choose the right field template to render the data (the `DisplayFormat` if used by itself uses the string template).

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

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

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

The following code shows combining attributes on one line:

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

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

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

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

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

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

Additional resources

- *Working with Forms*
- *Globalization and localization*
- *Introduction to Tag Helpers*
- *Authoring Tag Helpers*

Examining the Details and Delete methods

By Rick Anderson

Open the Movie controller and examine the `Details` method:

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
```

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

The MVC scaffolding engine that created this action method adds a comment showing a HTTP request that invokes the method. In this case it's a GET request with three URL segments, the `Movies` controller, the `Details` method and a `id` value. Recall these segments are defined in Startup.

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

Code First makes it easy to search for data using the `SingleOrDefaultAsync` method. An important security feature built into the method is that the code verifies that the search method has found a movie before the code tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from *http://localhost:xxxx/Movies/Details/1* to something like *http://localhost:xxxx/Movies/Details/12345* (or some other value that doesn't represent an actual movie). If you did not check for a null movie, the app would throw an exception.

Examine the `Delete` and `DeleteConfirmed` methods.

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}
```

Note that the `HTTP GET Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

The `[HttpPost]` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)


// POST: Movies/Delete/
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two `Delete` methods – one for GET and one for POST – that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the `DeleteConfirmed` method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the `notUsed` parameter. You could do the same thing here for the `[HttpPost] Delete` method:

```
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
{
    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}
```

1.3.5 ASP.NET Core on Nano Server

By Sourabh Shirhatti

> : This tutorial uses a pre-release version of the Nano Server installation option of Windows Server Technical Preview 5. You may use the software in the virtual hard disk image only to internally demonstrate and evaluate it. You may not use the software in a live operating environment. Please see https://go.microsoft.com/fwlink/?LinkId=624232 for specific information about the end date for the preview.

In this tutorial, you'll take an existing ASP.NET Core app and deploy it to a Nano Server instance running IIS.

Sections:

Introduction

Nano Server is an installation option in Windows Server 2016, offering a tiny footprint, better security and better servicing than Server Core or full Server. Please consult the official Nano Server documentation for more details. There are 3 ways for you try out Nano Server for yourself:

1. You can download the Windows Server 2016 Technical Preview 5 ISO file, and build a Nano Server image

2. Download the Nano Server developer VHD

3. Create a VM in Azure using the Nano Server image in the Azure Gallery. If you don't have an Azure account, you can get a free 30-day trial

In this tutorial, we will be using the pre-built Nano Server Developer VHD from Windows Server Technical Preview 5.

Before proceeding with this tutorial, you will need the *published* output of an existing ASP.NET Core application. Ensure your application is built to run in a **64-bit** process.

Setting up the Nano Server Instance

Create a new Virtual Machine using Hyper-V on your development machine using the previously downloaded VHD. The machine will require you to set an administator password before logging on. At the VM console, press F11 to set the password before the first logon.

After setting the local password, you will manage Nano Server using PowerShell remoting.

Connecting to your Nano Server Instance using PowerShell Remoting

Open an elevated PowerShell window to add your remote Nano Server instance to your `TrustedHosts` list.

```
$nanoServerIpAddress = "10.83.181.14"
Set-Item WSMan:\localhost\Client\TrustedHosts "$nanoServerIpAddress" –Concatenate –Force
```

`NOTE:` Replace the variable `$nanoServerIpAddress` with the correct IP address.

Once you have added your Nano Server instance to your `TrustedHosts`, you can connect to it using PowerShell remoting

```
$nanoServerSession = New-PSSession –ComputerName $nanoServerIpAddress –Credential ~\Administrator
Enter-PSSession $nanoServerSession
```

A successful connection results in a prompt with a format looking like: `[10.83.181.14]: PS C:\Users\Administrator\Documents>`

Creating a file share

Create a file share on the Nano server so that the published application can be copied to it. Run the following commands in the remote session:

```
mkdir C:\PublishedApps\AspNetCoreSampleForNano
netsh advfirewall firewall set rule group="File and Printer Sharing" new enable=yes
net share AspNetCoreSampleForNano=c:\PublishedApps\AspNetCoreSampleForNano /GRANT:EVERYONE`,FULL
```

After running the above commands you should be able to access this share by visiting \\<nanoserver-ip-address>\AspNetCoreSampleForNano in the host machine's Windows Explorer.

Open port in the Firewall

Run the following commands in the remote session to open up a port in the firewall to listen for TCP traffic.

```
New-NetFirewallRule -Name "AspNet5 IIS" -DisplayName "Allow HTTP on TCP/8000" -Protocol TCP -LocalPo
```

Installing IIS

Add the `NanoServerPackage` provider from the PowerShell gallery. Once the provider is installed and imported, you can install Windows packages.

Run the following commands in the PowerShell session that was created earlier:

```
Install-PackageProvider NanoServerPackage
Import-PackageProvider NanoServerPackage
Install-NanoServerPackage -Name Microsoft-NanoServer-IIS-Package
```

To quickly verify if IIS is setup correctly, you can visit the url `http://<nanoserver-ip-address>/` and should see a welcome page. When IIS is installed, by default a web site called `Default Web Site` listening on port 80 is created.

Installing the ASP.NET Core Module (ANCM)

The ASP.NET Core Module is an IIS 7.5+ module which is responsible for process management of ASP.NET Core HTTP listeners and to proxy requests to processes that it manages. At the moment, the process to install the ASP.NET Core Module for IIS is manual. You will need to install the version of the .NET Core Windows Server Hosting bundle on a regular (not Nano) machine. After installing the bundle on a regular machine, you will need to copy the following files to the file share that we created earlier.

On a regular (not Nano) machine run the following copy commands:

```
copy C:\windows\system32\inetsrv\aspnetcore.dll ``\\<nanoserver-ip-address>\AspNetCoreSampleForNano``
copy C:\windows\system32\inetsrv\config\schema\aspnetcore_schema.xml ``\\<nanoserver-ip-address>\AspN
```

On a Nano machine, you will need to copy the following files from the file share that we created earlier to the valid locations. So, run the following copy commands:

```
copy C:\PublishedApps\AspNetCoreSampleForNano\aspnetcore.dll C:\windows\system32\inetsrv\
copy C:\PublishedApps\AspNetCoreSampleForNano\aspnetcore_schema.xml C:\windows\system32\inetsrv\conf
```

Run the following script in the remote session:

```
# Backup existing applicationHost.config
copy C:\Windows\System32\inetsrv\config\applicationHost.config C:\Windows\System32\inetsrv\config\app

Import-Module IISAdministration

# Initialize variables
$aspNetCoreHandlerFilePath="C:\windows\system32\inetsrv\aspnetcore.dll"
Reset-IISServerManager -confirm:$false
$sm = Get-IISServerManager

# Add AppSettings section
$sm.GetApplicationHostConfiguration().RootSectionGroup.Sections.Add("appSettings")

# Set Allow for handlers section
$appHostconfig = $sm.GetApplicationHostConfiguration()
$section = $appHostconfig.GetSection("system.webServer/handlers")
$section.OverrideMode="Allow"

# Add aspNetCore section to system.webServer
$sectionaspNetCore = $appHostConfig.RootSectionGroup.SectionGroups["system.webServer"].Sections.Add('
$sectionaspNetCore.OverrideModeDefault = "Allow"
$sm.CommitChanges()

# Configure globalModule
Reset-IISServerManager -confirm:$false
$globalModules = Get-IISConfigSection "system.webServer/globalModules" | Get-IISConfigCollection
New-IISConfigCollectionElement $globalModules -ConfigAttribute @{"name"="AspNetCoreModule";"image"=$a

# Configure module
$modules = Get-IISConfigSection "system.webServer/modules" | Get-IISConfigCollection
New-IISConfigCollectionElement $modules -ConfigAttribute @{"name"="AspNetCoreModule"}

# Backup existing applicationHost.config
copy C:\Windows\System32\inetsrv\config\applicationHost.config C:\Windows\System32\inetsrv\config\app
```

NOTE: Delete the files `aspnetcore.dll` and `aspnetcore_schema.xml` from the share after the above step.

Installing .NET Core Framework

If you published a portable app, .NET Core must be installed on the target machine. Execute the following Powershell script in a remote Powershell session to install the .NET Framework on your Nano Server.

```
$SourcePath = "https://dotnetcli.blob.core.windows.net/dotnet/beta/Binaries/Latest/dotnet-win-x64.lat
$DestinationPath = "C:\dotnet"

$EditionId = (Get-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion' -Name 'Edit

if (($EditionId -eq "ServerStandardNano") -or
    ($EditionId -eq "ServerDataCenterNano") -or
    ($EditionId -eq "NanoServer") -or
    ($EditionId -eq "ServerTuva")) {

    $TempPath = [System.IO.Path]::GetTempFileName()
    if (($SourcePath -as [System.URI]).AbsoluteURI -ne $null)
    {
        $handler = New-Object System.Net.Http.HttpClientHandler
```

```powershell
        $client = New-Object System.Net.Http.HttpClient($handler)
        $client.Timeout = New-Object System.TimeSpan(0, 30, 0)
        $cancelTokenSource = [System.Threading.CancellationTokenSource]::new()
        $responseMsg = $client.GetAsync([System.Uri]::new($SourcePath), $cancelTokenSource.Token)
        $responseMsg.Wait()
        if (!$responseMsg.IsCanceled)
        {
            $response = $responseMsg.Result
            if ($response.IsSuccessStatusCode)
            {
                $downloadedFileStream = [System.IO.FileStream]::new($TempPath, [System.IO.FileMode]::
                $copyStreamOp = $response.Content.CopyToAsync($downloadedFileStream)
                $copyStreamOp.Wait()
                $downloadedFileStream.Close()
                if ($copyStreamOp.Exception -ne $null)
                {
                    throw $copyStreamOp.Exception
                }
            }
        }
    }
    else
    {
        throw "Cannot copy from $SourcePath"
    }
    [System.IO.Compression.ZipFile]::ExtractToDirectory($TempPath, $DestinationPath)
    Remove-Item $TempPath
}
```

Publishing the application

Copy over the published output of your existing application to the file share.

You may need to make changes to your *web.config* to point to where you extracted dotnet.exe. Alternatively, you can add dotnet.exe to your path.

Example of how a web.config might look like if dotnet.exe was **not** on the path:

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified"
    </handlers>
    <aspNetCore processPath="C:\dotnet\dotnet.exe" arguments=".\AspNetCoreSampleForNano.dll" stdoutLo
  </system.webServer>
</configuration>
```

Run the following commands in the remote session to create a new site in IIS for the published app. This script uses the DefaultAppPool for simplicity. For more considerations on running under an application pool, see *Application Pools*.

```powershell
Import-module IISAdministration
New-IISSite -Name "AspNetCore" -PhysicalPath c:\PublishedApps\AspNetCoreSampleForNano -BindingInforma
```

Running the Application

The published web app should be accessible in browser at `http://<nanoserver-ip-address>:8000`. If you have set up logging as described in *Log creation and redirection*, you should be able to view your logs at *C:\PublishedApps\AspNetCoreSampleForNano\logs*.

1.3.6 Creating Backend Services for Native Mobile Applications

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.3.7 Developing ASP.NET Core applications using dotnet watch

By Victor Hurdugaci

Introduction

`dotnet watch` is a development time tool that runs a `dotnet` command when source files change. It can be used to compile, run tests, or publish when code changes.

In this tutorial we'll use an existing WebApi application that calculates the sum and product of two numbers to demonstrate the use cases of `dotnet watch`. The sample application contains an intentional bug that we'll fix as part of this tutorial.

Getting started

Start by downloading the sample application. It contains two projects, `WebApp` (a web application) and `WebAppTests` (unit tests for the web application)

In a console, open the folder where you downloaded the sample application and run:

1. `dotnet restore`
2. `cd WebApp`
3. `dotnet run`

The console output will show messages similar to the ones below, indicating that the application is now running and waiting for requests:

```
$ dotnet run
Project WebApp (.NETCoreApp,Version=v1.0) will be compiled because inputs were modified
Compiling WebApp for .NETCoreApp,Version=v1.0

Compilation succeeded.
  0 Warning(s)
  0 Error(s)
```

```
Time elapsed 00:00:02.6049991

Hosting environment: Production
Content root path: /Users/user/dev/aspnet/Docs/aspnet/tutorials/dotnet-watch/sample/WebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

In a web browser, navigate to `http://localhost:5000/api/math/sum?a=4&b=5` and you should see the
result `9`.

If you navigate to `http://localhost:5000/api/math/product?a=4&b=5` instead, you'd expect to get
the result `20`. Instead, you get `9` again.

We'll fix that.

Adding `dotnet watch` to a project

1. Add `Microsoft.DotNet.Watcher.Tools` to the `tools` section of the *WebApp/project.json* file as in
   the example below:

```
"tools": {
  "Microsoft.DotNet.Watcher.Tools": "1.0.0-preview2-final"
},
```

2. Run `dotnet restore`.

The console output will show messages similar to the ones below:

```
log  : Restoring packages for /Users/user/dev/aspnet/Docs/aspnet/tutorials/dotnet-watch/sample/WebApp
log  : Restoring packages for tool 'Microsoft.DotNet.Watcher.Tools' in /Users/user/dev/aspnet/Docs/as
log  : Installing Microsoft.DotNet.Watcher.Core 1.0.0-preview2-final.
log  : Installing Microsoft.DotNet.Watcher.Tools 1.0.0-preview2-final.
```

Running `dotnet` commands using `dotnet watch`

Any `dotnet` command can be run with `dotnet watch`: For example:

Command	Command with watch
`dotnet run`	`dotnet watch run`
`dotnet run -f net451`	`dotnet watch run -f net451`
`dotnet run -f net451 -- --arg1`	`dotnet watch run -f net451 -- --arg1`
`dotnet test`	`dotnet watch test`

To run `WebApp` using the watcher, run `dotnet watch run` in the `WebApp` folder. The console output will show
messages similar to the ones below, indicating that `dotnet watch` is now watching code files:

```
user$ dotnet watch run
[DotNetWatcher] info: Running dotnet with the following arguments: run
[DotNetWatcher] info: dotnet process id: 39746
Project WebApp (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
Hosting environment: Production
Content root path: /Users/user/dev/aspnet/Docs/aspnet/tutorials/dotnet-watch/sample/WebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Making changes with `dotnet watch`

Make sure `dotnet watch` is running.

Let's fix the bug that we discovered when we tried to compute the product of two number.

Open *WebApp/Controllers/MathController.cs*.

We've intentionally introduced a bug in the code.

```
public static int Product(int a, int b)
{
    // We have an intentional bug here
    // + should be *
    return a + b;
}
```

Fix the code by replacing `a + b` with `a * b`.

Save the file. The console output will show messages similar to the ones below, indicating that `dotnet watch` detected a file change and restarted the application.

```
[DotNetWatcher] info: File changed: /Users/user/dev/aspnet/Docs/aspnet/tutorials/dotnet-watch/sample,
[DotNetWatcher] info: Running dotnet with the following arguments: run
[DotNetWatcher] info: dotnet process id: 39940
Project WebApp (.NETCoreApp,Version=v1.0) will be compiled because inputs were modified
Compiling WebApp for .NETCoreApp,Version=v1.0
Compilation succeeded.
  0 Warning(s)
  0 Error(s)
Time elapsed 00:00:03.3312829

Hosting environment: Production
Content root path: /Users/user/dev/aspnet/Docs/aspnet/tutorials/dotnet-watch/sample/WebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Verify `http://localhost:5000/api/math/product?a=4&b=5` returns the correct result.

Running tests using `dotnet watch`

The file watcher can run other `dotnet` commands like `test` or `publish`.

1. Open the `WebAppTests` folder that already has `dotnet watch` in *project.json*.

2. Run `dotnet watch test`.

If you previously fixed the bug in the `MathController` then you'll see an output similar to the one below, otherwise you'll see a test failure:

```
WebAppTests user$ dotnet watch test
[DotNetWatcher] info: Running dotnet with the following arguments: test
[DotNetWatcher] info: dotnet process id: 40193
Project WebApp (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
Project WebAppTests (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
xUnit.net .NET CLI test runner (64-bit .NET Core osx.10.11-x64)
  Discovering: WebAppTests
  Discovered:  WebAppTests
  Starting:    WebAppTests
  Finished:    WebAppTests
```

```
=== TEST EXECUTION SUMMARY ===
   WebAppTests  Total: 2, Errors: 0, Failed: 0, Skipped: 0, Time: 0.259s
SUMMARY: Total: 1 targets, Passed: 1, Failed: 0.
[DotNetWatcher] info: dotnet exit code: 0
[DotNetWatcher] info: Waiting for a file to change before restarting dotnet...
```

Once all the tests run, the watcher will indicate that it's waiting for a file to change before restarting `dotnet test`.

3. Open the controller file in *WebApp/Controllers/MathController.cs* and change some code. If you haven't fixed the product bug, do it now. Save the file.

`dotnet watch` will detect the file change and rerun the tests. The console output will show messages similar to the one below:

```
[DotNetWatcher] info: File changed: /Users/user/dev/aspnet/Docs/aspnet/tutorials/dotnet-watch/sample/
[DotNetWatcher] info: Running dotnet with the following arguments: test
[DotNetWatcher] info: dotnet process id: 40233
Project WebApp (.NETCoreApp,Version=v1.0) will be compiled because inputs were modified
Compiling WebApp for .NETCoreApp,Version=v1.0
Compilation succeeded.
  0 Warning(s)
  0 Error(s)
Time elapsed 00:00:03.2127590
Project WebAppTests (.NETCoreApp,Version=v1.0) will be compiled because dependencies changed
Compiling WebAppTests for .NETCoreApp,Version=v1.0
Compilation succeeded.
  0 Warning(s)
  0 Error(s)
Time elapsed 00:00:02.1204052

xUnit.net .NET CLI test runner (64-bit .NET Core osx.10.11-x64)
  Discovering: WebAppTests
  Discovered:  WebAppTests
  Starting:    WebAppTests
  Finished:    WebAppTests
=== TEST EXECUTION SUMMARY ===
   WebAppTests  Total: 2, Errors: 0, Failed: 0, Skipped: 0, Time: 0.260s
SUMMARY: Total: 1 targets, Passed: 1, Failed: 0.
[DotNetWatcher] info: dotnet exit code: 0
[DotNetWatcher] info: Waiting for a file to change before restarting dotnet...
```

1.4 Fundamentals

1.4.1 Application Startup

By Steve Smith

ASP.NET Core provides complete control of how individual requests are handled by your application. The `Startup` class is the entry point to the application, setting up configuration and wiring up services the application will use. Developers configure a request pipeline in the `Startup` class that is used to handle all requests made to the application.

Sections:

- *The Startup class*
- *The Configure method*
- *The ConfigureServices method*
- *Services Available in Startup*
- *Additional Resources*

The Startup class

In ASP.NET Core, the `Startup` class provides the entry point for an application, and is required for all applications. It's possible to have environment-specific startup classes and methods (see *Working with Multiple Environments*), but regardless, one `Startup` class will serve as the entry point for the application. ASP.NET searches the primary assembly for a class named `Startup` (in any namespace). You can specify a different assembly to search using the *Hosting:Application* configuration key. It doesn't matter whether the class is defined as `public`; ASP.NET will still load it if it conforms to the naming convention. If there are multiple `Startup` classes, this will not trigger an exception. ASP.NET will select one based on its namespace (matching the project's root namespace first, otherwise using the class in the alphabetically first namespace).

The `Startup` class can optionally accept dependencies in its constructor that are provided through *dependency injection*. Typically, the way an application will be configured is defined within its Startup class's constructor (see *Configuration*). The Startup class must define a `Configure` method, and may optionally also define a `ConfigureServices` method, which will be called when the application is started.

The Configure method

The `Configure` method is used to specify how the ASP.NET application will respond to individual HTTP requests. At its simplest, you can configure every request to receive the same response. However, most real-world applications require more functionality than this. More complex sets of pipeline configuration can be encapsulated in *middleware* and added using extension methods on IApplicationBuilder.

Your `Configure` method must accept an IApplicationBuilder parameter. Additional services, like `IHostingEnvironment` and `ILoggerFactory` may also be specified, in which case these services will be *injected* by the server if they are available. In the following example from the default web site template, you can see several extension methods are used to configure the pipeline with support for BrowserLink, error pages, static files, ASP.NET MVC, and Identity.

```
1  public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
2  {
3      loggerFactory.AddConsole(Configuration.GetSection("Logging"));
4      loggerFactory.AddDebug();
5
6      if (env.IsDevelopment())
7      {
8          app.UseDeveloperExceptionPage();
9          app.UseDatabaseErrorPage();
10         app.UseBrowserLink();
11     }
12     else
13     {
14         app.UseExceptionHandler("/Home/Error");
15     }
16
17     app.UseStaticFiles();
18
```

```
19        app.UseIdentity();
20
21        // Add external authentication middleware below. To configure them please see http://go.microsof
22
23        app.UseMvc(routes =>
24        {
25            routes.MapRoute(
26                name: "default",
27                template: "{controller=Home}/{action=Index}/{id?}");
28        });
29    }
```

Each `Use` extension method adds *middleware* to the request pipeline. For instance, the `UseMvc` extension method adds the *routing* middleware to the request pipeline and configures *MVC* as the default handler.

You can learn all about middleware and using IApplicationBuilder to define your request pipeline in the *Middleware* topic.

The ConfigureServices method

Your `Startup` class can optionally include a `ConfigureServices` method for configuring services that are used by your application. The `ConfigureServices` method is a public method on your `Startup` class that takes an IServiceCollection instance as a parameter and optionally returns an `IServiceProvider`. The `ConfigureServices` method is called before `Configure`. This is important, because some features like ASP.NET MVC require certain services to be added in `ConfigureServices` before they can be wired up to the request pipeline.

Just as with `Configure`, it is recommended that features that require substantial setup within `ConfigureServices` be wrapped up in extension methods on IServiceCollection. You can see in this example from the default web site template that several `Add[Something]` extension methods are used to configure the app to use services from Entity Framework, Identity, and MVC:

```
1   public void ConfigureServices(IServiceCollection services)
2   {
3       // Add framework services.
4       services.AddDbContext<ApplicationDbContext>(options =>
5           options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
6
7       services.AddIdentity<ApplicationUser, IdentityRole>()
8           .AddEntityFrameworkStores<ApplicationDbContext>()
9           .AddDefaultTokenProviders();
10
11      services.AddMvc();
12
13      // Add application services.
14      services.AddTransient<IEmailSender, AuthMessageSender>();
15      services.AddTransient<ISmsSender, AuthMessageSender>();
16  }
```

Adding services to the services container makes them available within your application via *dependency injection*. Just as the `Startup` class is able to specify dependencies its methods require as parameters, rather than hard-coding to a specific implementation, so too can your middleware, MVC controllers and other classes in your application.

The `ConfigureServices` method is also where you should add configuration option classes that you would like to have available in your application. See the *Configuration* topic to learn more about configuring options.

Services Available in Startup

ASP.NET Core provides certain application services and objects during your application's startup. You can request certain sets of these services by simply including the appropriate interface as a parameter on your `Startup` class's constructor or one of its `Configure` or `ConfigureServices` methods. The services available to each method in the `Startup` class are described below. The framework services and objects include:

IApplicationBuilder Used to build the application request pipeline. Available only to the `Configure` method in `Startup`. Learn more about *Request Features*.

IHostingEnvironment Provides the current `EnvironmentName`, `ContentRootPath`, `WebRootPath`, and web root file provider. Available to the `Startup` constructor and `Configure` method.

ILoggerFactory Provides a mechanism for creating loggers. Available to the `Startup` constructor and `Configure` method. Learn more about *Logging*.

IServiceCollection The current set of services configured in the container. Available only to the `ConfigureServices` method, and used by that method to configure the services available to an application.

Looking at each method in the `Startup` class in the order in which they are called, the following services may be requested as parameters:

Startup Constructor - `IApplicationEnvironment` - `IHostingEnvironment` - `ILoggerFactory`

ConfigureServices - `IServiceCollection`

Configure - `IApplicationBuilder` - `IApplicationEnvironment` - `IHostingEnvironment` - `ILoggerFactory`

: Although `ILoggerFactory` is available in the constructor, it is typically configured in the `Configure` method. Learn more about *Logging*.

Additional Resources

- *Working with Multiple Environments*
- *Middleware*
- *Open Web Interface for .NET (OWIN)*

1.4.2 Middleware

By Steve Smith and Rick Anderson

Sections:
- *What is middleware*
- *Creating a middleware pipeline with IApplicationBuilder*
- *Built-in middleware*
- *Writing middleware*
- *Additional Resources*

View or download sample code

What is middleware

Middleware are software components that are assembled into an application pipeline to handle requests and responses. Each component chooses whether to pass the request on to the next component in the pipeline, and can perform certain actions before and after the next component is invoked in the pipeline. Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

Request delegates are configured using Run, Map, and Use extension methods on the IApplicationBuilder type that is passed into the `Configure` method in the `Startup` class. An individual request delegate can be specified in-line as an anonymous method, or it can be defined in a reusable class. These reusable classes are *middleware*, or *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline, or short-circuiting the chain if appropriate.

Migrating HTTP Modules to Middleware explains the difference between request pipelines in ASP.NET Core and the previous versions and provides more middleware samples.

Creating a middleware pipeline with IApplicationBuilder

The ASP.NET request pipeline consists of a sequence of request delegates, called one after the next, as this diagram shows (the thread of execution follows the black arrows):

Each delegate has the opportunity to perform operations before and after the next delegate. Any delegate can choose to stop passing the request on to the next delegate, and instead handle the request itself. This is referred to as short-circuiting the request pipeline, and is desirable because it allows unnecessary work to be avoided. For example, an authorization middleware might only call the next delegate if the request is authenticated; otherwise it could short-circuit the pipeline and return a "Not Authorized" response. Exception handling delegates need to be called early on in the pipeline, so they are able to catch exceptions that occur in deeper calls within the pipeline.

You can see an example of setting up the request pipeline in the default web site template that ships with Visual Studio 2015. The `Configure` method adds the following middleware components:

1. Error handling (for both development and non-development environments)

2. Static file server

3. Authentication

4. MVC

```
1  public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
2  {
3      loggerFactory.AddConsole(Configuration.GetSection("Logging"));
4      loggerFactory.AddDebug();
5
6      if (env.IsDevelopment())
7      {
8          app.UseDeveloperExceptionPage();
9          app.UseDatabaseErrorPage();
10         app.UseBrowserLink();
11     }
12     else
13     {
14         app.UseExceptionHandler("/Home/Error");
15     }
16
17     app.UseStaticFiles();
18
19     app.UseIdentity();
20
21     // Add external authentication middleware below. To configure them please see http://go.microsoft
22
23     app.UseMvc(routes =>
24     {
25         routes.MapRoute(
26             name: "default",
27             template: "{controller=Home}/{action=Index}/{id?}");
28     });
29  }
```

In the code above (in non-development environments), UseExceptionHandler is the first middleware added to the pipeline, therefore will catch any exceptions that occur in later calls.

The static file module provides no authorization checks. Any files served by it, including those under *wwwroot* are publicly available. If you want to serve files based on authorization:

1. Store them outside of *wwwroot* and any directory accessible to the static file middleware.

2. Deliver them through a controller action, returning a FileResult where authorization is applied.

A request that is handled by the static file module will short circuit the pipeline. (see *Working with Static Files*.) If the request is not handled by the static file module, it's passed on to the Identity module, which performs authentication. If the request is not authenticated, the pipeline is short circuited. If the request does not fail authentication, the last stage of this pipeline is called, which is the MVC framework.

: The order in which you add middleware components is generally the order in which they take effect on the request, and then in reverse for the response. This can be critical to your app's security, performance and functionality. In the code above, the static file middleware is called early in the pipeline so it can handle requests and short circuit without going through unnecessary components. The authentication middleware is added to the pipeline before anything that handles requests that need to be authenticated. Exception handling must be registered before other middleware components in order to catch exceptions thrown by those components.

The simplest possible ASP.NET application sets up a single request delegate that handles all requests. In this case, there isn't really a request "pipeline", so much as a single anonymous function that is called in response to every HTTP request.

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

The first App.Run delegate terminates the pipeline. In the following example, only the first delegate ("Hello, World!") will run.

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

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

You chain multiple request delegates together; the next parameter represents the next delegate in the pipeline. You can terminate (short-circuit) the pipeline by *not* calling the *next* parameter. You can typically perform actions both before and after the next delegate, as this example demonstrates:

```
public void ConfigureLogInline(IApplicationBuilder app, ILoggerFactory loggerfactory)
{
    loggerfactory.AddConsole(minLevel: LogLevel.Information);
    var logger = loggerfactory.CreateLogger(_environment);
    app.Use(async (context, next) =>
    {
        logger.LogInformation("Handling request.");
        await next.Invoke();
        logger.LogInformation("Finished handling request.");
    });

    app.Run(async context =>
    {
        await context.Response.WriteAsync("Hello from " + _environment);
    });
}
```

> **:** Avoid modifying HttpResponse after invoking next, one of the next components in the pipeline may have written to the response, causing it to be sent to the client.

: This ConfigureLogInline method is called when the application is run with an environment set to LogInline. Learn more about *Working with Multiple Environments*. We will be using variations of Configure[Environment] to show different options in the rest of this article. The easiest way to run the samples in Visual Studio is with the web command, which is configured in *project.json*. See also *Application Startup*.

In the above example, the call to await next.Invoke() will call into the next delegate await context.Response.WriteAsync("Hello from " + _environment);. The client will receive the expected response ("Hello from LogInline"), and the server's console output includes both the before and after messages:

Run, Map, and Use

You configure the HTTP pipeline using Run, Map, and Use. The `Run` method short circuits the pipeline (that is, it will not call a `next` request delegate). Thus, `Run` should only be called at the end of your pipeline. `Run` is a convention, and some middleware components may expose their own Run[Middleware] methods that should only run at the end of the pipeline. The following two middleware are equivalent as the `Use` version doesn't use the `next` parameter:

```csharp
public void ConfigureEnvironmentOne(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Hello from " + _environment);
    });
}

public void ConfigureEnvironmentTwo(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Hello from " + _environment);
    });
}
```

: The IApplicationBuilder interface exposes a single `Use` method, so technically they're not all *extension* methods.

We've already seen several examples of how to build a request pipeline with `Use`. `Map*` extensions are used as a convention for branching the pipeline. The current implementation supports branching based on the request's path, or using a predicate. The `Map` extension method is used to match request delegates based on a request's path. `Map` simply accepts a path and a function that configures a separate middleware pipeline. In the following example, any request with the base path of `/maptest` will be handled by the pipeline configured in the `HandleMapTest` method.

```
private static void HandleMapTest(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Map Test Successful");
    });
}


public void ConfigureMapping(IApplicationBuilder app)
{
    app.Map("/maptest", HandleMapTest);


}
```

: When `Map` is used, the matched path segment(s) are removed from `HttpRequest.Path` and appended to `HttpRequest.PathBase` for each request.

In addition to path-based mapping, the `MapWhen` method supports predicate-based middleware branching, allowing separate pipelines to be constructed in a very flexible fashion. Any predicate of type `Func<HttpContext, bool>` can be used to map requests to a new branch of the pipeline. In the following example, a simple predicate is used to detect the presence of a query string variable `branch`:

```
private static void HandleBranch(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Branch used.");
    });
}


public void ConfigureMapWhen(IApplicationBuilder app)
{
    app.MapWhen(context => {
        return context.Request.Query.ContainsKey("branch");
    }, HandleBranch);

    app.Run(async context =>
    {
        await context.Response.WriteAsync("Hello from " + _environment);
    });
}
```

Using the configuration shown above, any request that includes a query string value for `branch` will use the pipeline defined in the `HandleBranch` method (in this case, a response of "Branch used."). All other requests (that do not define a query string value for `branch`) will be handled by the delegate defined on line 17.

You can also nest Maps:

```
app.Map("/level1", level1App => {
    level1App.Map("/level2a", level2AApp => {
        // "/level1/level2a"
        //...
    });
    level1App.Map("/level2b", level2BApp => {
        // "/level1/level2b"
        //...
    });
```

```
});
```

Built-in middleware

ASP.NET ships with the following middleware components:

1.1: Middleware

Middleware	Description
Authentication	Provides authentication support.
CORS	Configures Cross-Origin Resource Sharing.
Routing	Define and constrain request routes.
Session	Provides support for managing user sessions.
Static Files	Provides support for serving static files, and directory browsing.

Writing middleware

The CodeLabs middleware tutorial provides a good introduction to writing middleware.

For more complex request handling functionality, the ASP.NET team recommends implementing the middleware in its own class, and exposing an `IApplicationBuilder` extension method that can be called from the `Configure` method. The simple logging middleware shown in the previous example can be converted into a middleware class that takes in the next `RequestDelegate` in its constructor and supports an `Invoke` method as shown:

1.1: RequestLoggerMiddleware.cs

```csharp
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;

namespace MiddlewareSample
{
    public class RequestLoggerMiddleware
    {
        private readonly RequestDelegate _next;
        private readonly ILogger _logger;

        public RequestLoggerMiddleware(RequestDelegate next, ILoggerFactory loggerFactory)
        {
            _next = next;
            _logger = loggerFactory.CreateLogger<RequestLoggerMiddleware>();
        }

        public async Task Invoke(HttpContext context)
        {
            _logger.LogInformation("Handling request: " + context.Request.Path);
            await _next.Invoke(context);
            _logger.LogInformation("Finished handling request.");
        }
    }
}
```

The middleware follows the Explicit Dependencies Principle and exposes all of its dependencies in its constructor. Middleware can take advantage of the UseMiddleware<T> extension to inject services directly into their constructors,

as shown in the example below. Dependency injected services are automatically filled, and the extension takes a `params` array of arguments to be used for non-injected parameters.

1.2: RequestLoggerExtensions.cs

```
public static class RequestLoggerExtensions
{
    public static IApplicationBuilder UseRequestLogger(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<RequestLoggerMiddleware>();
    }
}
```

Using the extension method and associated middleware class, the `Configure` method becomes very simple and readable.

```
public void ConfigureLogMiddleware(IApplicationBuilder app,
    ILoggerFactory loggerfactory)
{
    loggerfactory.AddConsole(minLevel: LogLevel.Information);

    app.UseRequestLogger();

    app.Run(async context =>
    {
        await context.Response.WriteAsync("Hello from " + _environment);
    });
}
```

Although `RequestLoggerMiddleware` requires an `ILoggerFactory` parameter in its constructor, neither the `Startup` class nor the `UseRequestLogger` extension method need to explicitly supply it. Instead, it is automatically provided through dependency injection performed within `UseMiddleware<T>`.

Testing the middleware (by setting the `Hosting:Environment` environment variable to `LogMiddleware`) should result in output like the following (when using WebListener):

: The UseStaticFiles extension method (which creates the StaticFileMiddleware) also uses `UseMiddleware<T>`. In this case, the `StaticFileOptions` parameter is passed in, but other constructor parameters are supplied by `UseMiddleware<T>` and dependency injection.

Additional Resources

- CodeLabs middleware tutorial
- Sample code used in this doc
- *Migrating HTTP Modules to Middleware*
- *Application Startup*
- *Request Features*

1.4.3 Working with Static Files

By Rick Anderson

Static files, such as HTML, CSS, image, and JavaScript, are assets that an ASP.NET Core app can serve directly to clients.

View or download sample code

Sections

- *Serving static files*
- *Static file authorization*
- *Enabling directory browsing*
- *Serving a default document*
- *UseFileServer*
- *Non-standard content types*
- *Additional Resources*

Serving static files

Static files are typically located in the `web root` (*<content-root>/wwwroot*) folder. See Content root and Web root in *Introduction to ASP.NET Core* for more information. You generally set the content root to be the current directory so that your project's `web root` will be found while in development.

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();

    host.Run();
}
```

Static files can be stored in any folder under the `web root` and accessed with a relative path to that root. For example, when you create a default Web application project using Visual Studio, there are several folders created within the *wwwroot* folder - *css*, *images*, and *js*. The URI to access an image in the *images* subfolder:

- http://<app>/images/<imageFileName>

- http://localhost:9189/images/banner3.svg

In order for static files to be served, you must configure the *Middleware* to add static files to the pipeline. The static file middleware can be configured by adding a dependency on the *Microsoft.AspNetCore.StaticFiles* package to your project and then calling the `UseStaticFiles` extension method from `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    app.UseStaticFiles();
}
```

`app.UseStaticFiles();` makes the files in `web root` (*wwwroot* by default) servable. Later I'll show how to make other directory contents servable with `UseStaticFiles`.

You must include "Microsoft.AspNetCore.StaticFiles" in the *project.json* file.

: `web root` defaults to the *wwwroot* directory, but you can set the `web root` directory with `UseWebRoot`. See *Introduction to ASP.NET Core* for more information.

Suppose you have a project hierarchy where the static files you wish to serve are outside the `web root`. For example:

- wwwroot

    - css

    - images

    - ...

- MyStaticFiles

    - test.png

For a request to access *test.png*, configure the static files middleware as follows:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    app.UseStaticFiles();

    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"MyStaticFiles")),
        RequestPath = new PathString("/StaticFiles")
    });
}
```

A request to `http://<app>/StaticFiles/test.png` will serve the *test.png* file.

Static file authorization

The static file module provides **no** authorization checks. Any files served by it, including those under *wwwroot* are publicly available. To serve files based on authorization:

- Store them outside of *wwwroot* and any directory accessible to the static file middleware **and**

• Serve them through a controller action, returning a `FileResult` where authorization is applied

Enabling directory browsing

Directory browsing allows the user of your web app to see a list of directories and files within a specified directory. Directory browsing is disabled by default for security reasons (see *Considerations*). To enable directory browsing, call the `UseDirectoryBrowser` extension method from `Startup.Configure`:

```csharp
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\images")),
        RequestPath = new PathString("/MyImages")
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\images")),
        RequestPath = new PathString("/MyImages")
    });
}
```

And add required services by calling `AddDirectoryBrowser` extension method from `Startup.ConfigureServices`:

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

The code above allows directory browsing of the *wwwroot/images* folder using the URL http://<app>/MyImages, with links to each file and folder:

See *Considerations* on the security risks when enabling browsing.

Note the two `app.UseStaticFiles` calls. The first one is required to serve the CSS, images and JavaScript in the *wwwroot* folder, and the second call for directory browsing of the *wwwroot/images* folder using the URL http://<app>/MyImages:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\images")),
        RequestPath = new PathString("/MyImages")
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\images")),
        RequestPath = new PathString("/MyImages")
    });
}
```

Serving a default document

Setting a default home page gives site visitors a place to start when visiting your site. In order for your Web app to serve a default page without the user having to fully qualify the URI, call the `UseDefaultFiles` extension method from `Startup.Configure` as follows.

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

: `UseDefaultFiles` must be called before `UseStaticFiles` to serve the default file. `UseDefaultFiles` is a URL re-writer that doesn't actually serve the file. You must enable the static file middleware (`UseStaticFiles`) to serve the file.

With `UseDefaultFiles`, requests to a folder will search for:

- default.htm
- default.html
- index.htm
- index.html

The first file found from the list will be served as if the request was the fully qualified URI (although the browser URL will continue to show the URI requested).

The following code shows how to change the default file name to *mydefault.html*.

```
public void Configure(IApplicationBuilder app)
{
    // Serve my app-specific default file, if present.
    DefaultFilesOptions options = new DefaultFilesOptions();
    options.DefaultFileNames.Clear();
    options.DefaultFileNames.Add("mydefault.html");
    app.UseDefaultFiles(options);
```

```
        app.UseStaticFiles();
}
```

UseFileServer

UseFileServer combines the functionality of UseStaticFiles, UseDefaultFiles, and UseDirectoryBrowser.

The following code enables static files and the default file to be served, but does not allow directory browsing:

```
app.UseFileServer();
```

The following code enables static files, default files and directory browsing:

```
app.UseFileServer(enableDirectoryBrowsing: true);
```

See *Considerations* on the security risks when enabling browsing. As with UseStaticFiles, UseDefaultFiles, and UseDirectoryBrowser, if you wish to serve files that exist outside the web root, you instantiate and configure an FileServerOptions object that you pass as a parameter to UseFileServer. For example, given the following directory hierarchy in your Web app:

- wwwroot

    – css

    – images

    – ...

- MyStaticFiles

    – test.png

    – default.html

Using the hierarchy example above, you might want to enable static files, default files, and browsing for the MyStaticFiles directory. In the following code snippet, that is accomplished with a single call to FileServerOptions.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    app.UseStaticFiles();

    app.UseFileServer(new FileServerOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"MyStaticFiles")),
        RequestPath = new PathString("/StaticFiles"),
        EnableDirectoryBrowsing = true
    });
}
```

If enableDirectoryBrowsing is set to true you are required to call AddDirectoryBrowser extension method from Startup.ConfigureServices:

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

Using the file hierarchy and code above:

URI	Response
http://<app>/StaticFiles/test.png	StaticFiles/test.png
http://<app>/StaticFiles	MyStaticFiles/default.html

If no default named files are in the *MyStaticFiles* directory, http://<app>/StaticFiles returns the directory listing with clickable links:

: `UseDefaultFiles` and `UseDirectoryBrowser` will take the url http://<app>/StaticFiles without the trailing slash and cause a client side redirect to http://<app>/StaticFiles/ (adding the trailing slash). Without the trailing slash relative URLs within the documents would be incorrect.

FileExtensionContentTypeProvider

The `FileExtensionContentTypeProvider` class contains a collection that maps file extensions to MIME content types. In the following sample, several file extensions are registered to known MIME types, the ".rtf" is replaced, and ".mp4" is removed.

```
public void Configure(IApplicationBuilder app)
{
    // Set up custom content types -associating file extension to MIME type
    var provider = new FileExtensionContentTypeProvider();
    // Add new mappings
    provider.Mappings[".myapp"] = "application/x-msdownload";
    provider.Mappings[".htm3"] = "text/html";
    provider.Mappings[".image"] = "image/png";
    // Replace an existing mapping
    provider.Mappings[".rtf"] = "application/x-msdownload";
    // Remove MP4 videos.
    provider.Mappings.Remove(".mp4");

    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\images")),
        RequestPath = new PathString("/MyImages"),
        ContentTypeProvider = provider
```

```
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\images")),
        RequestPath = new PathString("/MyImages")
    });
}
```

See MIME content types.

Non-standard content types

The ASP.NET static file middleware understands almost 400 known file content types. If the user requests a file of an unknown file type, the static file middleware returns a HTTP 404 (Not found) response. If directory browsing is enabled, a link to the file will be displayed, but the URI will return an HTTP 404 error.

The following code enables serving unknown types and will render the unknown file as an image.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    app.UseStaticFiles(new StaticFileOptions
    {
        ServeUnknownFileTypes = true,
        DefaultContentType = "image/png"
    });
}
```

With the code above, a request for a file with an unknown content type will be returned as an image.

> : Enabling `ServeUnknownFileTypes` is a security risk and using it is discouraged. `FileExtensionContentTypeProvider` (explained below) provides a safer alternative to serving files with non-standard extensions.

Considerations

> : `UseDirectoryBrowser` and `UseStaticFiles` can leak secrets. We recommend that you **not** enable directory browsing in production. Be careful about which directories you enable with `UseStaticFiles` or `UseDirectoryBrowser` as the entire directory and all sub-directories will be accessible. We recommend keeping public content in its own directory such as *<content root>/wwwroot*, away from application views, configuration files, etc.

- The URLs for content exposed with `UseDirectoryBrowser` and `UseStaticFiles` are subject to the case sensitivity and character restrictions of their underlying file system. For example, Windows is case insensitive, but Mac and Linux are not.

- ASP.NET Core applications hosted in IIS use the ASP.NET Core Module to forward all requests to the application including requests for static files. The IIS static file handler is not used because it doesn't get a chance to handle requests before they are handled by the ASP.NET Core Module.

- To remove the IIS static file handler (at the server or website level):

    - Navigate to the **Modules** feature

    - Select **StaticFileModule** in the list

  – Tap **Remove** in the **Actions** sidebar

> **:** If the IIS static file handler is enabled **and** the ASP.NET Core Module (ANCM) is not correctly configured (for example if *web.config* was not deployed), static files will be served.

- Code files (including c# and Razor) should be placed outside of the app project's `web root` (*wwwroot* by default). This creates a clean separation between your app's client side content and server side source code, which prevents server side code from being leaked.

Additional Resources

- *Middleware*
- *Introduction to ASP.NET Core*

1.4.4 Routing

By Steve Smith

Routing middleware is used to map requests to route handlers. Routes are configured when the application starts up, and can extract values from the URL that will be passed as arguments to route handlers. Routing functionality is also responsible for generating links that correspond to routes in ASP.NET apps.

> **Sections**
>
> - *Routing Middleware*
> - *Configuring Routing*
> - *Template Routes*
> - *Route Builder Extensions*
> - *Link Generation*
> - *Recommendations*

View or download sample code

Routing Middleware

The routing *middleware* uses *routes* to map requests to an `IRouter` instance. The `IRouter` instance chooses whether or not to handle the request, and how. The request is considered handled if its `RouteContext.Handler` property is set to a non-null value. If no route handler is found for a request, then the middleware calls *next* (and the next middleware in the request pipeline is invoked).

To use routing, add it to the **dependencies** in *project.json*:

```
"dependencies": {
  "Microsoft.NETCore.App": {
    "version": "1.0.0-rc2-3002702",
    "type": "platform"
  },
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0-rc2-final",
  "Microsoft.AspNetCore.Routing": "1.0.0-rc2-final",
  "Microsoft.Extensions.Logging.Console": "1.0.0-rc2-final",
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0-rc2-final"
},
```

Add routing to `ConfigureServices` in *Startup.cs*:

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

Configuring Routing

Routing is enabled in the `Configure` method in the `Startup` class. Create an instance of `RouteBuilder`, passing a reference to `IApplicationBuilder`. You can optionally provide a `DefaultHandler` as well. Add additional routes using `MapRoute` and when finished call `app.UseRouter`.

```
public void Configure(IApplicationBuilder app,
    ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(minLevel: LogLevel.Trace);

    var defaultHandler = new RouteHandler((c) =>
        c.Response.WriteAsync($"Hello world! Route values: " +
        $"{string.Join(", ", c.GetRouteData().Values)}")
        );

    var routeBuilder = new RouteBuilder(app, defaultHandler);

    routeBuilder.AddHelloRoute(app);

    routeBuilder.MapRoute(
        "Track Package Route",
        "package/{operation:regex(track|create|detonate)}/{id:int}");

    app.UseRouter(routeBuilder.Build());
```

Pass `UseRouter` the result of the `RouteBuilder.Build` method.

: If you are only configuring a single route, you can simply call `app.UseRouter` and pass in the `IRouter` instance you wish to use, bypassing the need to use a `RouteBuilder`.

The `defaultHandler` route handler is used as the default for the `RouteBuilder`. Calls to `MapRoute` will use this handler by default. A second handler is configured within the `HelloRouter` instance added by the `AddHelloRoute` extension method. This extension methods adds a new `Route` to the `RouteBuilder`, passing in an instance of `IRouter`, a template string, and an `IInlineConstraintResolver` (which is responsible for enforcing any route constraints specified):

```
public static IRouteBuilder AddHelloRoute(this IRouteBuilder routeBuilder,
    IApplicationBuilder app)
{
    routeBuilder.Routes.Add(new Route(new HelloRouter(),
        "hello/{name:alpha}",
        app.ApplicationServices.GetService<IInlineConstraintResolver>()));

    return routeBuilder;
```

`HelloRouter` is a custom `IRouter` implementation. `AddHelloRoute` adds an instance of this router to the `RouteBuilder` using a template string, "hello/{name:alpha}". This template will only match requests of the form

"hello/{name}" where *name* is constrained to be alphabetical. Matching requests will be handled by `HelloRouter` (which implements the `IRouter` interface), which responds to requests with a simple greeting.

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;

namespace RoutingSample
{
    public class HelloRouter : IRouter
    {
        public Task RouteAsync(RouteContext context)
        {
            var name = context.RouteData.Values["name"] as string;
            if (String.IsNullOrEmpty(name))
            {
                return Task.FromResult(0);
            }
            var requestPath = context.HttpContext.Request.Path;
            if (requestPath.StartsWithSegments("/hello", StringComparison.OrdinalIgnoreCase))
            {
                context.Handler = async c =>
                {
                    await c.Response.WriteAsync($"Hi, {name}!");
                };
            }
            return Task.FromResult(0);
        }

        public VirtualPathData GetVirtualPath(VirtualPathContext context)
        {
            return null;
        }
    }
}
```

`HelloRouter` checks to see if `RouteData` includes a value for the key `name`. If not, it immediately returns without handling the request. Likewise, it checks to see if the request begins with "/hello". Otherwise, the `Handler` property is set to a delegate that responds with a greeting. Setting the `Handler` property prevents additional routes from handling the request. The `GetVirtualPath` method is used for *link generation*.

: Remember, it's possible for a particular route **template** to match a given request, but the associated route **handler** can still reject it, allowing a different route to handle the request.)

This route was configured to use an *inline constraint*, signified by the `:alpha` in the name route value. This constraint limits which requests this route will handle, in this case to alphabetical values for `name`. Thus, a request for "/hello/steve" will be handled, but a request to "/hello/123" will not (instead, in this sample the request will not match any routes and will use the "app.Run" delegate).

Template Routes

The most common way to define routes is using `TemplateRoute` and route template strings. When a `TemplateRoute` matches, it calls its target `IRouter` handler. In a typical MVC app, you might use a default template route with a string like this one:

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

This route template would be handled by the `MvcRouteHandler` `IRouter` instance. Tokens within curly braces (`{ }`) define *route value* parameters which will be bound if the route is matched. You can define more than one route value parameter in a route segment, but they must be separated by a literal value. For example `{controller=Home}{action=Index}` would not be a valid route, since there is no literal value between `{controller}` and `{action}`. These route value parameters must have a name, and may have additional attributes specified.

You can use the `*` character as a prefix to a route value name to bind to the rest of the URI. For example, `blog/{*slug}` would match any URI that started with `/blog/` and had any value following it (which would be assigned to the `slug` route value).

Route value parameters may have *default values*, designated by specifying the default after the parameter name, separated by an =. For example, `controller=Home` would define `Home` as the default value for `controller`. The default value is used if no value is present in the URL for the parameter. In addition to default values, route parameters may be optional (specified by appending a `?` to the end of the parameter name, as in `id?`). The difference between optional and "has default" is that a route parameter with a default value always produces a value; an optional parameter may not. Route parameters may also have constraints, which further restrict which routes the template will match.

The following table demonstrates some route template and their expected behavior.

1.2: Route Template Values

Route Template	Example Matching URL	Notes
hello	/hello	Will only match the single path '/hello'
{Page=Home}	/	Will match and set `Page` to `Home`.
{Page=Home}	/Contact	Will match and set `Page` to `Contact`
{controller}/{action}/{id?}	/Products/List	Will map to `Products` controller and `List` method; Since `id` was not supplied in the URL, it's ignored.
{controller}/{action}/{id?}	/Products/Details/123	Will map to `Products` controller and `Details` method, with `id` set to `123`.
{controller=Home}/{action=Index}/{id?}	/	Will map to `Home` controller and `Index` method; `id` is ignored.

Route Constraints

Adding a colon `:` after the name allows additional inline constraints to be set on a route value parameter. Constraints with types always use the invariant culture - they assume the URL is non-localizable. Route constraints limit which URLs will match a route - URLs that do not match the constraint are ignored by the route.

1.3: Inline Route Constraints

constraint	Example	Example Match	Notes
`int`	{id:int}	123	Matches any integer
`bool`	{active:bool}	true	Matches `true` or `false`
`datetime`	{dob:datetime}	2016-01-01	Matches a valid `DateTime` value (in the invariant culture - see options)
`decimal`	{price:decimal}	49.99	Matches a valid `decimal` value
`double`	{price:double}	4.234	Matches a valid `double` value
`float`	{price:float}	3.14	Matches a valid `float` value
`guid`	{id:guid}	7342570B-44E7-471C-A267-947DD2A35BF9	Matches a valid `Guid` value
`long`	{ticks:long}	123456789	Matches a valid `long` value
`minlength(value)`	{user-name:minlength(5)}	steve	String must be at least 5 characters long.
`maxlength(value)`	{file-name:maxlength(8)}	somefile	String must be no more than 8 characters long.
`length(min,max)`	{file-name:length(4,16)}	Somefile.txt	String must be at least 8 and no more than 16 characters long.
`min(value)`	{age:min(18)}	19	Value must be at least 18.
`max(value)`	{age:max(120)}	91	Value must be no more than 120.
`range(min,max)`	{age:range(18,120)}	91	Value must be at least 18 but no more than 120.
`alpha`	{name:alpha}	Steve	String must consist of alphabetical characters.
`regex(expression)`	{ssn:regex(d{3}-d{2}-d{4})}	123-45-6789	String must match the provided regular expression.
`required`	{name:required}	Steve	Used to enforce that a non-parameter value is present during during URL generation.

Inline constraints must match one of the above options, or an exception will be thrown.

: To constrain a parameter to a known set of possible values, you can use a regex: `{action:regex(list|get|create)}`. This would only match the `action` route value to `list`, `get`, or `create`. If passed into the constraints dictionary, the string "list|get|create" would be equivalent. Constraints that are passed in the constraints dictionary (not inline within a template) that don't match one of the known constraints are also treated as regular expressions.

: Avoid using constraints for **validation**, because doing so means that invalid input will result in a 404 (Not Found) instead of a 400 with an appropriate error message. Route constraints should be used to **disambiguate** between routes, not validate the inputs for a particular route.

Constraints can be *chained*. You can specify that a route value is of a certain type and also must fall within a specified range, for example: `{age:int:range(1,120)}`. Numeric constraints like `min`, `max`, and `range` will automatically convert the value to `long` before being applied unless another numeric type is specified.

Route templates must be unambiguous, or they will be ignored. For example, `{id?}/{foo}` is ambiguous, because it's not clear which route value would be bound to a request for "/bar". Similarly, `{*everything}/{plusone}` would be ambiguous, because the first route parameter would match everything from that part of the request on, so it's not clear what the `plusone` parameter would match.

: There is a special case route for filenames, such that you can define a route value like `files/{filename}.{ext?}`. When both `filename` and `ext` exist, both values will be populated. However, if only `filename` exists in the URL, the trailing period `.` is also optional. Thus, these would both match:

`/files/foo.txt` and `/files/foo`.

: Enable *Logging* to see how the built in routing implementations, such as `TemplateRoute`, match requests.

Route Builder Extensions

Several extension methods on RouteBuilder are available for convenience. The most common of these is `MapRoute`, which allows the specification of a route given a name and template, and optionally default values, constraints, and/or *data tokens*. When using these extensions, you must have specified the `DefaultHandler` and `ServiceProvider` properties of the `RouteBuilder` instance to which you're adding the route. These `MapRoute` extensions add new `TemplateRoute` instances to the `RouteBuilder` that each target the `IRouter` configured as the `DefaultHandler`.

: `MapRoute` doesn't take an `IRouter` parameter - it only adds routes that will be handled by the `DefaultHandler`. Since the default handler is an `IRouter`, it may decide not to handle the request. For example, MVC is typically configured as a default handler that only handles requests that match an available controller action.

Data Tokens

Data tokens represent data that is carried along if the route matches. They're implemented as a property bag for developer-specified data. You can use data tokens to store data you want to associate with a route, when you don't want the semantics of defaults. Data tokens have no impact on the **behavior** of the route, while defaults do. Data tokens can also be any arbitrary types, while defaults really need to be things that can be converted to/from strings.

Link Generation

Routing is also used to generate URLs based on route definitions. This is used by helpers to generate links to known actions on MVC *controllers*, but can also be used independent of MVC. Given a set of route values, and optionally a route name, you can produce a `VirtualPathContext` object. Using the `VirtualPathContext` object along with a `RouteCollection`, you can generate a `VirtualPath`. `IRouter` implementations participate in link generation through the `GetVirtualPath` method.

: Learn more about UrlHelper and *Routing to Controller Actions*.

The example below shows how to generate a link to a route given a dictionary of route values and a `RouteCollection`.

```
// demonstrate link generation
var trackingRouteCollection = new RouteCollection();
trackingRouteCollection.Add(routeBuilder.Routes[1]); // "Track Package Route"

app.Run(async (context) =>
{
    var dictionary = new RouteValueDictionary
    {
        {"operation","create" },
        {"id",123}
```

```
    };

    var vpc = new VirtualPathContext(context,
        null, dictionary, "Track Package Route");

    context.Response.ContentType = "text/html";
    await context.Response.WriteAsync("Menu<hr/>");
    await context.Response.WriteAsync(@"<a href='" +
        trackingRouteCollection.GetVirtualPath(vpc).VirtualPath +
        "'>Create Package 123</a><br/>");
```

The `VirtualPath` generated at the end of the sample above is `/package/create/123`.

The second parameter to the `VirtualPathContext` constructor is a collection of *ambient values*. Ambient values provide convenience by limiting the number of values a developer must specify within a certain request context. The current route values of the current request are considered ambient values for link generation. For example, in an MVC application if you are in the About action of the HomeController, you don't need to specify the controller route value to link to the Index action (the ambient value of Home will be used).

Ambient values that don't match a parameter are ignored, and ambient values are also ignored when an explicitly-provided value overrides it, going from left to right in the URL.

Values that are explicitly provided but which don't match anything are added to the query string.

<div align="center">1.4: Generating Links</div>

Matched Route	Ambient Values	Explicit Values	Result
`{controller}/{action}/{id?}`	controller="Home"	action="About"	`/Home/About`
`{controller}/{action}/{id?}`	controller="Home"	controller="Order",action="About"	`/Order/About`
`{controller}/{action}/{id?}`	controller="Home",color="Red"	action="About"	`/Home/About`
`{controller}/{action}/{id?}`	controller="Home"	action="About",color="Red"	`/Home/About?color=Red`

If a route has a default value that doesn't match a parameter and that value is explicitly provided, it must match the default value. For example:

```
routes.MapRoute("blog_route", "blog/{*slug}",
  defaults: new { controller = "Blog", action = "ReadPost" });
```

Link generation would only generate a link for this route when the matching values for controller and action are provided.

Recommendations

Routing is a powerful feature that is built into the default ASP.NET MVC project template such that most apps will be able to leverage it without having to customize its behavior. This is by design; customizing routing behavior is an advanced development approach. Keep in mind the following recommendations with regard to routing:

- Most apps shouldn't need custom routes. The default route will work in most cases.

- Attribute routes should be used for all APIs.

- Attribute routes are recommended for when you need complete control over your app's URLs.

- Conventional routing is recommended for when **all** of your controllers/actions fit a uniform URL convention.

- Don't use custom routes unless you understand them well and are sure you need them.

- Routes can be tricky to test and debug.

- Routes should not be used as a means of securing your controllers or their action methods.

- Avoid building or changing route collections at runtime.

1.4.5 Error Handling

By Steve Smith

When errors occur in your ASP.NET app, you can handle them in a variety of ways, as described in this article.

Sections

- *Configuring an Exception Handling Page*
- *Using the Developer Exception Page*
- *Configuring Status Code Pages*
- *Limitations of Exception Handling During Client-Server Interaction*
- *Server Exception Handling*
- *Startup Exception Handling*
- *ASP.NET MVC Error Handling*

View or download sample code

Configuring an Exception Handling Page

You configure the pipeline for each request in the `Startup` class's `Configure()` method (learn more about *Application Startup*). You can add a simple exception page, meant only for use during development, very easily. All that's required is to add a dependency on `Microsoft.AspNetCore.Diagnostics` to the project and then add one line to `Configure()` in `Startup.cs`:

```csharp
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env)
{
    app.UseIISPlatformHandler();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
```

The above code includes a check to ensure the environment is development before adding the call to `UseDeveloperExceptionPage`. This is a good practice, since you typically do not want to share detailed exception information about your application publicly while it is in production. *Learn more about configuring environments*.

The sample application includes a simple mechanism for creating an exception:

```csharp
public static void HomePage(IApplicationBuilder app)
{
    app.Run(async (context) =>
    {
        if (context.Request.Query.ContainsKey("throw"))
        {
            throw new Exception("Exception triggered!");
        }
        var builder = new StringBuilder();
```

```
        builder.AppendLine("<html><body>Hello World!");
        builder.AppendLine("<ul>");
        builder.AppendLine("<li><a href=\"/?throw=true\">Throw Exception</a></li>");
        builder.AppendLine("<li><a href=\"/missingpage\">Missing Page</a></li>");
        builder.AppendLine("</ul>");
        builder.AppendLine("</body></html>");

        context.Response.ContentType = "text/html";
        await context.Response.WriteAsync(builder.ToString());
    });
}
```

If a request includes a non-empty querystring parameter for the variable `throw` (e.g. a path of `/?throw=true`), an exception will be thrown. If the environment is set to `Development`, the developer exception page is displayed:

When not in development, it's a good idea to configure an exception handler path using the `UseExceptionHandler` middleware:

```
app.UseExceptionHandler("/Error");
```

Using the Developer Exception Page

The developer exception page displays useful diagnostics information when an unhandled exception occurs within the web processing pipeline. The page includes several tabs with information about the exception that was triggered and the request that was made. The first tab includes a stack trace:

The next tab shows the query string parameters, if any:

In this case, you can see the value of the `throw` parameter that was passed to this request. This request didn't have any cookies, but if it did, they would appear on the Cookies tab. You can see the headers that were passed in the last tab:

Configuring Status Code Pages

By default, your app will not provide a rich status code page for HTTP status codes such as 500 (Internal Server Error) or 404 (Not Found). You can configure the `StatusCodePagesMiddleware` adding this line to the `Configure` method:

```
app.UseStatusCodePages();
```

By default, this middleware adds very simple, text-only handlers for common status codes. For example, the following is the result of a 404 Not Found status code:

The middleware supports several different extension methods. You can pass it a custom lamdba expression:

```
app.UseStatusCodePages(context =>
    context.HttpContext.Response.SendAsync("Handler, status code: " +
    context.HttpContext.Response.StatusCode, "text/plain"));
```

Alternately, you can simply pass it a content type and a format string:

```
app.UseStatusCodePages("text/plain", "Response, status code: {0}");
```

The middleware can handle redirects (with either relative or absolute URL paths), passing the status code as part of the URL:

```
app.UseStatusCodePagesWithRedirects("~/errors/{0}");
```

In the above case, the client browser will see a `302 / Found` status and will redirect to the URL provided.

Alternately, the middleware can re-execute the request from a new path format string:

```
app.UseStatusCodePagesWithReExecute("/errors/{0}");
```

The `UseStatusCodePagesWithReExecute` method will still return the original status code to the browser, but will also execute the handler given at the path specified.

If you need to disable status code pages for certain requests, you can do so using the following code:

```
var statusCodePagesFeature = context.Features.Get<IStatusCodePagesFeature>();
if (statusCodePagesFeature != null)
{
    statusCodePagesFeature.Enabled = false;
}
```

Limitations of Exception Handling During Client-Server Interaction

Web apps have certain limitations to their exception handling capabilities because of the nature of disconnected HTTP requests and responses. Keep these in mind as you design your app's exception handling behavior.

1. Once the headers for a response have been sent, you cannot change the response's status code, nor can any exception pages or handlers run. The response must be completed or the connection aborted.

2. If the client disconnects mid-response, you cannot send them the rest of the content of that response.

3. There is always the possibility of an exception occuring one layer below your exception handling layer.

4. Don't forget, exception handling pages can have exceptions, too. It's often a good idea for production error pages to consist of purely static content.

Following the above recommendations will help ensure your app remains responsive and is able to gracefully handle exceptions that may occur.

Server Exception Handling

In addition to the exception handling logic in your app, the server hosting your app will perform some exception handling. If the server catches an exception before the headers have been sent it will send a 500 Internal Server Error response with no body. If it catches an exception after the headers have been sent it must close the connection. Requests that are not handled by your app will be handled by the server, and any exception that occurs will be handled by the server's exception handling. Any custom error pages or exception handling middleware or filters you have configured for your app will not affect this behavior.

Startup Exception Handling

One of the trickiest places to handle exceptions in your app is during its startup. Only the hosting layer can handle exceptions that take place during app startup. Exceptions that occur in your app's startup can also impact server behavior. For example, to enable SSL in Kestrel, one must configure the server with `KestrelServerOptions.UseHttps()`. If an exception happens before this line in `Startup`, then by default hosting will catch the exception, start the server, and display an error page on the non-SSL port. If an exception happens after that line executes, then the error page will be served over HTTPS instead.

ASP.NET MVC Error Handling

MVC apps have some additional options when it comes to handling errors, such as configuring exception filters and performing model validation.

Exception Filters

Exception filters can be configured globally or on a per-controller or per-action basis in an *MVC* app. These filters handle any unhandled exception that occurs during the execution of a controller action or another filter, and are not called otherwise. Exception filters are detailed in *filters*.

: Exception filters are good for trapping exceptions that occur within MVC actions, but they're not as flexible as error handling middleware. Prefer middleware for the general case, and use filters only where you need to do error handling *differently* based on which MVC action was chosen.

Handling Model State Errors

Model validation occurs prior to each controller action being invoked, and it is the action method's responsibility to inspect `ModelState.IsValid` and react appropriately. In many cases, the appropriate reaction is to return some kind of error response, ideally detailing the reason why model validation failed.

Some apps will choose to follow a standard convention for dealing with model validation errors, in which case a *filter* may be an appropriate place to implement such a policy. You should test how your actions behave with valid and invalid model states (learn more about *testing controller logic*).

1.4.6 Globalization and localization

Rick Anderson, Damien Bowden, Bart Calixto, Nadeem Afana

Creating a multilingual website with ASP.NET Core will allow your site to reach a wider audience. ASP.NET Core provides services and middleware for localizing into different languages and cultures.

Internationalization involves Globalization and Localization. Globalization is the process of designing apps that support different cultures. Globalization adds support for input, display, and output of a defined set of language scripts that relate to specific geographic areas.

Localization is the process of adapting a globalized app, which you have already processed for localizability, to a particular culture/locale. For more information see **Globalization and localization terms** near the end of this document.

App localization involves the following:

1. Make the app's content localizable

2. Provide localized resources for the languages and cultures you support

3. Implement a strategy to select the language/culture for each request

---

Sections:

- *Make the app's content localizable*
- *View localization*
- *DataAnnotations localization*
- *Provide localized resources for the languages and cultures you support*
- *Working with resource files*
- *Implement a strategy to select the language/culture for each request*
- *Resource file naming*
- *Globalization and localization terms*
- *Additional Resources*

Make the app's content localizable

Introduced in ASP.NET Core, `IStringLocalizer` and `IStringLocalizer<T>` were architected to improve productivity when developing localized apps. `IStringLocalizer` uses the ResourceManager and ResourceReader to provide culture-specific resources at run time. The simple interface has an indexer and an `IEnumerable` for returning localized strings. `IStringLocalizer` doesn't require you to store the default language strings in a resource file. You can develop an app targeted for localization and not need to create resource files early in development. The code below shows how to wrap the string "About Title" for localization.

```
using Microsoft.AspNet.Mvc;
using Microsoft.Extensions.Localization;

namespace Localization.StarterWeb.Controllers
{
    [Route("api/[controller]")]
    public class AboutController : Controller
    {
        private readonly IStringLocalizer<AboutController> _localizer;

        public AboutController(IStringLocalizer<AboutController> localizer)
        {
            _localizer = localizer;
        }

        [HttpGet]
        public string Get()
        {
            return _localizer["About Title"];
        }
    }
}
```

In the code above, the `IStringLocalizer<T>` implementation comes from *Dependency Injection*. If the localized value of "About Title" is not found, then the indexer key is returned, that is, the string "About Title". You can leave the default language literal strings in the app and wrap them in the localizer, so that you can focus on developing the app. You develop your app with your default language and prepare it for the localization step without first creating a default resource file. Alternatively, you can use the traditional approach and provide a key to retrieve the default language string. For many developers the new workflow of not having a default language *.resx* file and simply wrapping the string literals can reduce the overhead of localizing an app. Other developers will prefer the traditional work flow as it can make it easier to work with longer string literals and make it easier to update localized strings.

Use the IHtmlLocalizer<T> implementation for resources that contain HTML. `IHtmlLocalizer` HTML encodes arguments that are formatted in the resource string, but not the resource string. In the sample highlighted below, only the value of `name` parameter is HTML encoded.

```
using System;
using Microsoft.AspNet.Http;
using Microsoft.AspNet.Localization;
using Microsoft.AspNet.Mvc;
using Microsoft.AspNet.Mvc.Localization;

namespace Localization.StarterWeb.Controllers
{
    public class BookController : Controller
    {
        private readonly IHtmlLocalizer<BookController> _localizer;

        public BookController(IHtmlLocalizer<BookController> localizer)
        {
            _localizer = localizer;
        }

        public IActionResult Hello(string name)
        {
            ViewData["Message"] = _localizer["<b>Hello</b><i> {0}</i>", name];

            return View();
```

```
        }
```

Note You generally want to only localize text and not HTML.

At the lowest level, you can get IStringLocalizerFactory out of *Dependency Injection*:

```
public class TestController : Controller
{
    private readonly IStringLocalizer _localizer;
    private readonly IStringLocalizer _localizer2;

    public TestController(IStringLocalizerFactory factory)
    {
        _localizer = factory.Create(typeof(SharedResource));
        _localizer2 = factory.Create("SharedResource", location: null);
    }

    public IActionResult About()
    {
        ViewData["Message"] = _localizer["Your application description page."]
            + " loc 2: " + _localizer2["Your application description page."];

        return View();
    }
```

The code above demonstrates each of the two factory create methods.

You can partition your localized strings by controller, area, or have just one container. In the sample app, a dummy class named SharedResource is used for shared resources.

```
// Dummy class to group shared resources

namespace Localization.StarterWeb
{
    public class SharedResource
    {
    }
}
```

Some developers use the Startup class to contain global or shared strings. In the sample below, the InfoController and the SharedResource localizers are used:

```
public class InfoController : Controller
{
    private readonly IStringLocalizer<InfoController> _localizer;
    private readonly IStringLocalizer<SharedResource> _sharedLocalizer;

    public InfoController(IStringLocalizer<InfoController> localizer,
                  IStringLocalizer<SharedResource> sharedLocalizer)
    {
        _localizer = localizer;
        _sharedLocalizer = sharedLocalizer;
    }

    public string TestLoc()
    {
        string msg = "Shared resx: " + _sharedLocalizer["Hello!"] +
                     " Info resx " + _localizer["Hello!"];
        return msg;
    }
```

View localization

The IViewLocalizer service provides localized strings for a view. The ViewLocalizer class implements this interface and finds the resource location from the view file path. The following code shows how to use the default implementation of `IViewLocalizer`:

```
@using Microsoft.AspNet.Mvc.Localization

@inject IViewLocalizer Localizer

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

<p>@Localizer["Use this area to provide additional information."]</p>
```

The default implementation of `IViewLocalizer` finds the resource file based on the view's file name. There is no option to use a global shared resource file. `ViewLocalizer` implements the localizer using IHtmlLocalizer, so Razor doesn't HTML encode the localized string. You can parameterize resource strings and `IViewLocalizer` will HTML encode the parameters, but not the resource string. Consider the following Razor markup:

```
@Localizer["<i>Hello</i> <b>{0}!</b>", UserManager.GetUserName(User)]
```

A French resource file could contain the following:

Key	Value
<i>Hello</i> {0}!	<i>Bonjour</i> {0}!

The rendered view would contain the HTML markup from the resource file.

> **Note** You generally want to only localize text and not HTML.

To use a shared resource file in a view, inject IHtmlLocalizer<T>:

```
@using Microsoft.AspNet.Mvc.Localization
@using Localization.StarterWeb.Services

@inject IViewLocalizer Localizer
@inject IHtmlLocalizer<SharedResource> SharedLocalizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>

<h1>@SharedLocalizer["Hello!"]</h1>
```

DataAnnotations localization

DataAnnotations error messages are localized with IStringLocalizer<T>. Using the option `ResourcesPath = "Resources"`, the error messages in `RegisterViewModel` can be stored in either of the following paths:

- Resources/ViewModels.Account.RegisterViewModel.fr.resx

- Resources/ViewModels/Account/RegisterViewModel.fr.resx

```
public class RegisterViewModel
{
    [Required(ErrorMessage = "The Email field is required.")]
    [EmailAddress(ErrorMessage = "The Email field is not a valid e-mail address.")]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "The Password field is required.")]
    [StringLength(8, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength =
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

The runtime doesn't look up localized strings for non-validation attributes. In the code above, "Email" (from `[Display(Name = "Email")]`) will not be localized.

Provide localized resources for the languages and cultures you support

SupportedCultures and SupportedUICultures

ASP.NET Core allows you to specify two culture values, `SupportedCultures` and `SupportedUICultures`. The [CultureInfo](#) object for `SupportedCultures` determines the results of culture-dependent functions, such as date, time, number, and currency formatting. `SupportedCultures` also determines the sorting order of text, casing conventions, and string comparisons. See [CultureInfo.CurrentCulture](#) for more info on how the server gets the Culture. The `SupportedUICultures` determines which translates strings (from *.resx* files) are looked up by the [ResourceManager](#). The `ResourceManager` simply looks up culture-specific strings that is determined by `CurrentUICulture`. Every thread in .NET has `CurrentCulture` and `CurrentUICulture` objects. ASP.NET Core inspects these values when rendering culture-dependent functions. For example, if the current thread's culture is set to "en-US" (English, United States), `DateTime.Now.ToLongDateString()` displays "Thursday, February 18, 2016", but if `CurrentCulture` is set to "es-ES" (Spanish, Spain) the output will be "jueves, 18 de febrero de 2016".

Working with resource files

A resource file is a useful mechanism for separating localizable strings from code. Translated strings for the non-default language are isolated *.resx* resource files. For example, you might want to create Spanish resource file named *Welcome.es.resx* containing translated strings. "es" is the language code for Spanish. To create this resource file in Visual Studio:

1. In **Solution Explorer**, right click on the folder which will contain the resource file > **Add** > **New Item**.

2. In the **Search installed templates** box, enter "resource" and name the file.

3. Enter the key value (native string) in the **Name** column and the translated string in the **Value** column.

Visual Studio shows the *Welcome.es.resx* file.

Generating resource files with Visual Studio

If you create a resource file in Visual Studio without a culture in the file name (for example, *Welcome.resx*), Visual Studio will create a C# class with a property for each string. That's usually not what you want with ASP.NET Core; you typically don't have a default *.resx* resource file (A *.resx* file without the culture name). We suggest you create the *.resx* file with a culture name (for example *Welcome.fr.resx*). When you create a *.resx* file with a culture name, Visual Studio will not generate the class file. We anticipate that many developers will **not** create a default language resource file.

Adding Other Cultures

Each language and culture combination (other than the default language) requires a unique resource file. You can create resource files for different cultures and locales by creating new resource files in which the ISO language codes are part of the file name (for example, **en-us**, **fr-ca**, and **en-gb**). These ISO codes are placed between the file name and the .resx file name extension, as in *Welcome.es-MX.resx* (Spanish/Mexico). To specify a culturally neutral language, you would eliminate the country code, such as *Welcome.fr.resx* for the French language.

Implement a strategy to select the language/culture for each request

Configuring localization

Localization is configured in the `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddLocalization(options => options.ResourcesPath = "Resources");

    services.AddMvc()
      .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
      .AddDataAnnotationsLocalization();
```

- `AddLocalization` Adds the localization services to the services container. The code above also sets the resources path to "Resources".

- `AddViewLocalization` Adds support for localized view files. In this sample view localization is based on the view file suffix. For example "fr" in the *Index.fr.cshtml* file.

- `AddDataAnnotationsLocalization` Adds support for localized `DataAnnotations` validation messages through `IStringLocalizer` abstractions.

Localization middleware

The current culture on a request is set in the localization *Middleware*. The localization middleware is enabled in the `Configure` method of *Startup.cs* file.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactor

    var supportedCultures = new[]
    {
        new CultureInfo("en-US"),
        new CultureInfo("en-AU"),
        new CultureInfo("en-GB"),
        new CultureInfo("en"),
        new CultureInfo("es-ES"),
        new CultureInfo("es-MX"),
        new CultureInfo("es"),
        new CultureInfo("fr-FR"),
        new CultureInfo("fr"),
    };

    app.UseRequestLocalization(new RequestLocalizationOptions
    {
        DefaultRequestCulture = new RequestCulture("en-US"),
        // Formatting numbers, dates, etc.
        SupportedCultures = supportedCultures,
        // UI strings that we have localized.
        SupportedUICultures = supportedCultures
    });

    // Remaining code omitted for brevity.
```

`UseRequestLocalization` initializes a `RequestLocalizationOptions` object. On every request the list of `RequestCultureProvider` in the `RequestLocalizationOptions` is enumerated and the first provider that can successfully determine the request culture is used. The default providers come from the `RequestLocalizationOptions` class:

1. QueryStringRequestCultureProvider

2. CookieRequestCultureProvider

3. AcceptLanguageHeaderRequestCultureProvider

The default list goes from most specific to least specific. Later in the article we'll see how you can change the order and even add a custom culture provider. If none of the providers can determine the request culture, the `DefaultRequestCulture` is used.

QueryStringRequestCultureProvider

Some apps will use a query string to set the culture and UI culture. For apps that use the cookie or Accept-Language header approach, adding a query string to the URL is useful for debugging and testing code. By default, the `QueryStringRequestCultureProvider` is registered as the first localization provider in the `RequestCultureProvider` list. You pass the query string parameters `culture` and `ui-culture`. The following example sets the specific culture (language and region) to Spanish/Mexico:

> http://localhost:5000/?culture=es-MX&ui-culture=es-MX

If you only pass in one of the two (`culture` or `ui-culture`), the query string provider will set both values using the one you passed in. For example, setting just the culture will set both the `Culture` and the `UICulture`:

> http://localhost:5000/?culture=es-MX

CookieRequestCultureProvider

Production apps will often provide a mechanism to set the culture with the ASP.NET Core culture cookie. Use the `MakeCookieValue` method to create a cookie.

The `CookieRequestCultureProvider DefaultCookieName` returns the default cookie name used to track the user's preferred culture information. The default cookie name is ".AspNetCore.Culture".

The cookie format is `c=%LANGCODE%|uic=%LANGCODE%`, where `c` is `Culture` and `uic` is `UICulture`, for example:

c='en-UK'|uic='en-US'

If you only specify one of culture info and UI culture, the specified culture will be used for both culture info and UI culture.

The Accept-Language HTTP header

The Accept-Language header is settable in most browsers and was originally intended to specify the user's language. This setting indicates what the browser has been set to send or has inherited from the underlying operating system. The Accept-Language HTTP header from a browser request is not an infallible way to detect the user's preferred language (see Setting language preferences in a browser). A production app should include a way for a user to customize their choice of culture.

Setting the Accept-Language HTTP header in IE

1. From the gear icon, tap **Internet Options**.

2. Tap **Languages**.

3. Tap **Set Language Preferences**.

4. Tap **Add a language**.

5. Add the language.

6. Tap the language, then tap **Move Up**.

Using a custom provider

Suppose you want to let your customers store their language and culture in your databases. You could write a provider to look up these values for the user. The following code shows how to add a custom provider:

```
services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[]
    {
        new CultureInfo("en-US"),
        new CultureInfo("fr")
    };
```

```
    options.DefaultRequestCulture = new RequestCulture(culture: "en-US", uiCulture: "en-US");
    options.SupportedCultures = supportedCultures;
    options.SupportedUICultures = supportedCultures;

    options.RequestCultureProviders.Insert(0, new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
      return new ProviderCultureResult("en");
    }));
});
```

Use `RequestLocalizationOptions` to add or remove localization providers.

Resource file naming

Resources are named for the type of their class minus the default namespace (which is also the name of the assembly). For example, a French resource in the `LocalizationWebsite.Web` project for the class `LocalizationWebsite.Web.Startup` would be named *Startup.fr.resx*. The class `LocalizationWebsite.Web.Controllers.HomeController` would be *Controllers.HomeController.fr.resx*. If for some reason your targeted class is not in the base namespace you will need the full type name. For example, in the sample project a type `ExtraNamespace.Tools` would be *ExtraNamespace.Tools.fr.resx*.

In the sample project, the `ConfigureServices` method sets the `ResourcesPath` to "Resources", so the project relative path for the home controller's French resource file is *Resources/Controllers.HomeController.fr.resx*. Alternatively, you can use folders to organize resource files. For the home controller, the path would be *Resources/Controllers/HomeController.fr.resx*. If you don't use the `ResourcesPath` option, the *.resx* file would go in the project base directory. The resource file for `HomeController` would be named *Controllers.HomeController.fr.resx*. The choice of using the dot or path naming convention depends on how you want to organize your resource files.

Resource name	Dot or path naming
Resources/Controllers.HomeController.fr.resx	Dot
Resources/Controllers/HomeController.fr.resx	Path

Resource files using `@inject IViewLocalizer` in Razor views follow a similar pattern. The resource file for a view can be named using either dot naming or path naming. Razor view resource files mimic the path of their associated view file. Assuming we set the `ResourcesPath` to "Resources", the French resource file associated with the *Views/Book/About.cshtml* view could be either of the following:

- Resources/Views/Home/About.fr.resx

- Resources/Views.Home.About.fr.resx

If you don't use the `ResourcesPath` option, the *.resx* file for a view would be located in the same folder as the view.

If you remove the ".fr" culture designator AND you have the culture set to French (via cookie or other mechanism), the default resource file is read and strings are localized. The Resource manager designates a default or fallback resource, when nothing meets your requested culture you're served the *.resx file without a culture designator. If you want to just return the key when missing a resource for the requested culture you must not have a default resource file.

Setting the culture programmatically

This sample **Localization.StarterWeb** project on GitHub contains UI to set the `Culture`. The *Views/Shared/_SelectLanguagePartial.cshtml* file allows you to select the culture from the list of supported cultures:

```
@using Microsoft.AspNet.Builder
@using Microsoft.AspNet.Http.Features
@using Microsoft.AspNet.Localization
@using Microsoft.AspNet.Mvc.Localization
@using Microsoft.Extensions.Options

@inject IViewLocalizer Localizer
@inject IOptions<RequestLocalizationOptions> LocOptions

@{
    var requestCulture = Context.Features.Get<IRequestCultureFeature>();
    var cultureItems = LocOptions.Value.SupportedUICultures
        .Select(c => new SelectListItem { Value = c.Name, Text = c.DisplayName })
        .ToList();
}

<div title="@Localizer["Request culture provider:"] @requestCulture?.Provider?.GetType().Name">
    <form id="selectLanguage" asp-controller="Home"
          asp-action="SetLanguage" asp-route-returnUrl="@Context.Request.Path"
          method="post" class="form-horizontal" role="form">
        @Localizer["Language:"] <select name="culture"
          asp-for="@requestCulture.RequestCulture.UICulture.Name" asp-items="cultureItems">
        </select>
    </form>
</div>
```

The *Views/Shared/_SelectLanguagePartial.cshtml* file is added to the `footer` section of the layout file so it will be available to all views:

```html
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <div class="row">
            <div class="col-md-6">
                <p>&copy; 2015 - Localization.StarterWeb</p>
            </div>
            <div class="col-md-6 text-right">
                @await Html.PartialAsync("_SelectLanguagePartial")
            </div>
        </div>
    </footer>
</div>
```

The `SetLanguage` method sets the culture cookie.

```csharp
[HttpPost]
public IActionResult SetLanguage(string culture, string returnUrl)
{
    Response.Cookies.Append(
        CookieRequestCultureProvider.DefaultCookieName,
        CookieRequestCultureProvider.MakeCookieValue(new RequestCulture(culture)),
        new CookieOptions { Expires = DateTimeOffset.UtcNow.AddYears(1) }
    );

    return LocalRedirect(returnUrl);
}
```

You can't simply plug in the *_SelectLanguagePartial.cshtml* to sample code for this project. The **Localization.StarterWeb** project on GitHub has code to flow the `RequestLocalizationOptions` to a Razor partial through the *Dependency Injection* container.

Globalization and localization terms

The process of localizing your app also requires a basic understanding of relevant character sets commonly used in modern software development and an understanding of the issues associated with them. Although all computers store text as numbers (codes), different systems store the same text using different numbers. The localization process refers to translating the app user interface (UI) for a specific culture/locale.

Localizability is an intermediate process for verifying that a globalized app is ready for localization.

The RFC 4646 format for the culture name is "<languagecode2>-<country/regioncode2>", where <languagecode2> is the language code and <country/regioncode2> is the subculture code. For example, `es-CL` for Spanish (Chile), `en-US` for English (United States), and `en-AU` for English (Australia). RFC 4646 is a combination of an ISO 639 two-letter lowercase culture code associated with a language and an ISO 3166 two-letter uppercase subculture code associated with a country or region. See Language Culture Name.

Internationalization is often abbreviated to "I18N". The abbreviation takes the first and last letters and the number of letters between them, so 18 stands for the number of letters between the first "I" and the last "N". The same applies to Globalization (G11N), and Localization (L10N).

Terms:

- Globalization (G11N): The process of making an app support different languages and regions.

- Localization (L10N): The process of customizing an app for a given language and region.

- Internationalization (I18N): Describes both globalization and localization.

- Culture: It is a language and, optionally, a region.

- Neutral culture: A culture that has a specified language, but not a region. (for example "en", "es")

- Specific culture: A culture that has a specified language and region. (for example "en-US", "en-GB", "es-CL")

- Locale: A locale is the same as a culture.

Additional Resources

- Localization.StarterWeb project used in the article.

- Resource Files in Visual Studio

- Resources in .resx Files

1.4.7 Configuration

Steve Smith, Daniel Roth

ASP.NET Core supports a variety of different configuration options. Application configuration data can come from files using built-in support for JSON, XML, and INI formats, as well as from environment variables, command line arguments or an in-memory collection. You can also write your own *custom configuration provider*.

Sections:

- *Getting and setting configuration settings*
- *Using the built-in sources*
- *Using Options and configuration objects*
- *Writing custom providers*
- *Summary*

View or download sample code

Getting and setting configuration settings

ASP.NET Core's configuration system has been re-architected from previous versions of ASP.NET, which relied on `System.Configuration` and XML configuration files like `web.config`. The new configuration model provides streamlined access to key/value based settings that can be retrieved from a variety of sources. Applications and frameworks can then access configured settings in a strongly typed fashion using the new *Options pattern*.

To work with settings in your ASP.NET application, it is recommended that you only instantiate a `Configuration` in your application's `Startup` class. Then, use the *Options pattern* to access individual settings.

At its simplest, `Configuration` is just a collection of sources, which provide the ability to read and write name/value pairs. If a name/value pair is written to `Configuration`, it is not persisted. This means that the written value will be lost when the sources are read again.

You must configure at least one source in order for `Configuration` to function correctly. The following sample shows how to test working with `Configuration` as a key/value store:

```
var builder = new ConfigurationBuilder();
builder.AddInMemoryCollection();
var config = builder.Build();
config["somekey"] = "somevalue";

// do some other work

var setting = config["somekey"]; // also returns "somevalue"
```

: You must set at least one configuration source.

It's not unusual to store configuration values in a hierarchical structure, especially when using external files (e.g. JSON, XML, INI). In this case, configuration values can be retrieved using a `:` separated key, starting from the root of the hierarchy. For example, consider the following *appsettings.json* file:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-WebApplication1-26e8893e-d7c
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

The application uses configuration to configure the right connection string. Access to the `DefaultConnection` setting is achieved through this key: `ConnectionStrings:DefaultConnection`, or by using the `GetConnectionString` extension method and passing in `"DefaultConnection"`.

The settings required by your application and the mechanism used to specify those settings (configuration being one example) can be decoupled using the *options pattern*. To use the options pattern you create your own options class (probably several different classes, corresponding to different cohesive groups of settings) that you can inject into your application using an options service. You can then specify your settings using configuration or whatever mechanism you choose.

: You could store your `Configuration` instance as a service, but this would unnecessarily couple your application to a single configuration system and specific configuration keys. Instead, you can use the *Options pattern* to avoid these issues.

Using the built-in sources

The configuration framework has built-in support for JSON, XML, and INI configuration files, as well as support for in-memory configuration (directly setting values in code) and the ability to pull configuration from environment variables and command line parameters. Developers are not limited to using a single configuration source. In fact several may be set up together such that a default configuration is overridden by settings from another source if they are present.

Adding support for additional configuration sources is accomplished through extension methods. These methods can be called on a `ConfigurationBuilder` instance in a standalone fashion, or chained together as a fluent API. Both of these approaches are demonstrated in the sample below.

```
// work with with a builder using multiple calls
var builder = new ConfigurationBuilder();
builder.SetBasePath(Directory.GetCurrentDirectory());
builder.AddJsonFile("appsettings.json").Build();
var connectionStringConfig = builder.Build();

// chain calls together as a fluent API
var config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json")
    .AddEntityFrameworkConfig(options =>
        options.UseSqlServer(connectionStringConfig.GetConnectionString("DefaultConnection"))
    )
    .Build();
```

The order in which configuration sources are specified is important, as this establishes the precedence with which settings will be applied if they exist in multiple locations. In the example below, if the same setting exists in both *appsettings.json* and in an environment variable, the setting from the environment variable will be the one that is used. The last configuration source specified "wins" if a setting exists in more than one location. The ASP.NET team recommends specifying environment variables last, so that the local environment can override anything set in deployed configuration files.

: To override nested keys through environment variables in shells that don't support : in variable names, replace them with __ (double underscore).

It can be useful to have environment-specific configuration files. This can be achieved using the following:

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);

    if (env.IsDevelopment())
    {
        // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID
        builder.AddUserSecrets();
    }

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

The `IHostingEnvironment` service is used to get the current environment. In the `Development` environment, the highlighted line of code above would look for a file named `appsettings.Development.json` and use its values, overriding any other values, if it's present. Learn more about *Working with Multiple Environments*.

When specifying files as configuration sources, you can optionally specify whether changes to the file should result in the settings being reloaded. This is configured by passing in a `true` value for the `reloadOnChange` parameter when calling `AddJsonFile` or similar file-based extension methods.

> **:** You should never store passwords or other sensitive data in configuration provider code or in plain text configuration files. You also shouldn't use production secrets in your development or test environments. Instead, such secrets should be specified outside the project tree, so they cannot be accidentally committed into the configuration provider repository. Learn more about *Working with Multiple Environments* and managing *Safe storage of app secrets during development*.

One way to leverage the order precedence of `Configuration` is to specify default values, which can be overridden. In the console application below, a default value for the `username` setting is specified in an in-memory collection, but this is overridden if a command line argument for `username` is passed to the application. You can see in the output how many different configuration sources are configured in the application at each stage of its execution.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Microsoft.Extensions.Configuration;
5
6  namespace ConfigConsole
7  {
8      public static class Program
9      {
10         public static void Main(string[] args)
11         {
12             var builder = new ConfigurationBuilder();
13             Console.WriteLine("Initial Config Sources: " + builder.Sources.Count());
14
15             builder.AddInMemoryCollection(new Dictionary<string, string>
16             {
17                 { "username", "Guest" }
18             });
19
20             Console.WriteLine("Added Memory Source. Sources: " + builder.Sources.Count());
21
```

```
22              builder.AddCommandLine(args);
23              Console.WriteLine("Added Command Line Source. Sources: " + builder.Sources.Count());
24
25              var config = builder.Build();
26              string username = config["username"];
27
28              Console.WriteLine($"Hello, {username}!");
29          }
30      }
31  }
```

When run, the program will display the default value unless a command line parameter overrides it.

```
C:\WINDOWS\system32\cmd.exe                                    —    □    ×

> dotnet run
Project ConfigConsole (.NETCoreApp,Version=v1.0) was previously compiled. Skipping
compilation.
Initial Config Sources: 0
Added Memory Source. Sources: 1
Added Command Line Source. Sources: 2
Hello, Guest!

> dotnet run --username "John Doe"
Project ConfigConsole (.NETCoreApp,Version=v1.0) was previously compiled. Skipping
compilation.
Initial Config Sources: 0
Added Memory Source. Sources: 1
Added Command Line Source. Sources: 2
Hello, John Doe!

>
```

Using Options and configuration objects

The options pattern enables using custom options classes to represent a group of related settings. A class needs to have a public read-write property for each setting and a constructor that does not take any parameters (e.g. a default constructor) in order to be used as an options class.

It's recommended that you create well-factored settings objects that correspond to certain features within your application, thus following the Interface Segregation Principle (ISP) (classes depend only on the configuration settings they use) as well as Separation of Concerns (settings for disparate parts of your app are managed separately, and thus are less likely to negatively impact one another).

A simple `MyOptions` class is shown here:

```csharp
public class MyOptions
{
    public string Option1 { get; set; }
    public int Option2 { get; set; }
```

```
}
```

Options can be injected into your application using the `IOptions<TOptions>` accessor service. For example, the following *controller* uses `IOptions<MyOptions>` to access the settings it needs to render the `Index` view:

```csharp
public class HomeController : Controller
{
    private readonly IOptions<MyOptions> _optionsAccessor;

    public HomeController(IOptions<MyOptions> optionsAccessor)
    {
        _optionsAccessor = optionsAccessor;
    }

    // GET: /<controller>/
    public IActionResult Index() => View(_optionsAccessor.Value);
}
```

: Learn more about *Dependency Injection*.

To setup the `IOptions<TOptions>` service you call the `AddOptions` extension method during startup in your `ConfigureServices` method:

```csharp
public void ConfigureServices(IServiceCollection services)
{
    // Setup options with DI
    services.AddOptions();
```

The `Index` view displays the configured options:

You configure options using the `Configure<TOptions>` extension method. You can configure options using a delegate or by binding your options to configuration:

```
public void ConfigureServices(IServiceCollection services)
{
    // Setup options with DI
    services.AddOptions();

    // Configure MyOptions using code
    services.Configure<MyOptions>(myOptions =>
    {
        myOptions.Option1 = "value1_from_action";
    });

    // Configure MyOptions using config by installing Microsoft.Extensions.Options.ConfigurationExten
    services.Configure<MyOptions>(Configuration);

    // Add framework services.
    services.AddMvc();
}
```

When you bind options to configuration, each property in your options type is bound to a configuration key of the form `property:subproperty:...`. For example, the `MyOptions.Option1` property is bound to the key `Option1`, which is read from the `option1` property in *appsettings.json*. Note that configuration keys are case insensitive.

Each call to `Configure<TOptions>` adds an `IConfigureOptions<TOptions>` service to the service container that is used by the `IOptions<TOptions>` service to provide the configured options to the application or framework. If you want to configure your options some other way (for example, reading settings

from a database) you can use the `AddSingleton<TOptions>` extension method to you specify a custom `IConfigureOptions<TOptions>` service directly.

You can have multiple `IConfigureOptions<TOptions>` services for the same option type and they are all applied in order. In the *example* above, the values of `Option1` and `Option2` are both specified in *appsettings.json*, but the value of `Option1` is overridden by the configured delegate with the value "value1_from_action".

Writing custom providers

In addition to using the built-in configuration providers, you can also write your own. To do so, you simply implement the `IConfigurationSource` interface, which exposes a `Build` method. The build method configures and returns an `IConfigurationProvider`.

Example: Entity Framework Settings

You may wish to store some of your application's settings in a database, and access them using Entity Framework Core (EF). There are many ways in which you could choose to store such values, ranging from a simple table with a column for the setting name and another column for the setting value, to having separate columns for each setting value. In this example, we're going to create a simple configuration provider that reads name-value pairs from a database using EF.

To start off we'll define a simple `ConfigurationValue` entity for storing configuration values in the database:

```
public class ConfigurationValue
{
    public string Id { get; set; }
    public string Value { get; set; }
}
```

You need a `ConfigurationContext` to store and access the configured values using EF:

```
public class ConfigurationContext : DbContext
{
    public ConfigurationContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<ConfigurationValue> Values { get; set; }
}
```

Create an `EntityFrameworkConfigurationSource` that inherits from `IConfigurationSource`:

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public class EntityFrameworkConfigurationSource : IConfigurationSource
    {
        private readonly Action<DbContextOptionsBuilder> _optionsAction;

        public EntityFrameworkConfigurationSource(Action<DbContextOptionsBuilder> optionsAction)
        {
            _optionsAction = optionsAction;
        }
```

```
        public IConfigurationProvider Build(IConfigurationBuilder builder)
        {
            return new EntityFrameworkConfigurationProvider(_optionsAction);
        }
    }
}
```

Next, create the custom configuration provider by inheriting from `ConfigurationProvider`. The configuration data is loaded by overriding the `Load` method, which reads in all of the configuration data from the configured database. For demonstration purposes, the configuration provider also takes care of initializing the database if it hasn't already been created and populated:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public class EntityFrameworkConfigurationProvider : ConfigurationProvider
    {
        public EntityFrameworkConfigurationProvider(Action<DbContextOptionsBuilder> optionsAction)
        {
            OptionsAction = optionsAction;
        }

        Action<DbContextOptionsBuilder> OptionsAction { get; }

        public override void Load()
        {
            var builder = new DbContextOptionsBuilder<ConfigurationContext>();
            OptionsAction(builder);

            using (var dbContext = new ConfigurationContext(builder.Options))
            {
                dbContext.Database.EnsureCreated();
                Data = !dbContext.Values.Any()
                    ? CreateAndSaveDefaultValues(dbContext)
                    : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
            }
        }

        private static IDictionary<string, string> CreateAndSaveDefaultValues(
            ConfigurationContext dbContext)
        {
            var configValues = new Dictionary<string, string>
                {
                    { "key1", "value_from_ef_1" },
                    { "key2", "value_from_ef_2" }
                };
            dbContext.Values.AddRange(configValues
                .Select(kvp => new ConfigurationValue { Id = kvp.Key, Value = kvp.Value })
                .ToArray());
            dbContext.SaveChanges();
            return configValues;
        }
    }
```

```
}
```

Note the values that are being stored in the database ("value_from_ef_1" and "value_from_ef_2"); these are displayed in the sample below to demonstrate the configuration is reading values from the database properly.

By convention you can also add an `AddEntityFrameworkConfiguration` extension method for adding the configuration source:

```csharp
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public static class EntityFrameworkExtensions
    {
        public static IConfigurationBuilder AddEntityFrameworkConfig(
            this IConfigurationBuilder builder, Action<DbContextOptionsBuilder> setup)
        {
            return builder.Add(new EntityFrameworkConfigurationSource(setup));
        }
    }
}
```

You can see an example of how to use this custom configuration provider in your application in the following example. Create a new `ConfigurationBuilder` to set up your configuration sources. To add the `EntityFrameworkConfigurationProvider`, you first need to specify the EF data provider and connection string. How should you configure the connection string? Using configuration of course! Add an *appsettings.json* file as a configuration source to bootstrap setting up the `EntityFrameworkConfigurationProvider`. By adding the database settings to an existing configuration with other sources specified, any settings specified in the database will override settings specified in *appsettings.json*:

```csharp
using System;
using System.IO;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public static class Program
    {
        public static void Main()
        {
            // work with with a builder using multiple calls
            var builder = new ConfigurationBuilder();
            builder.SetBasePath(Directory.GetCurrentDirectory());
            builder.AddJsonFile("appsettings.json").Build();
            var connectionStringConfig = builder.Build();

            // chain calls together as a fluent API
            var config = new ConfigurationBuilder()
                .SetBasePath(Directory.GetCurrentDirectory())
                .AddJsonFile("appsettings.json")
                .AddEntityFrameworkConfig(options =>
                    options.UseSqlServer(connectionStringConfig.GetConnectionString("DefaultConnectio
                )
                .Build();
```

```
            Console.WriteLine("key1={0}", config["key1"]);
            Console.WriteLine("key2={0}", config["key2"]);
            Console.WriteLine("key3={0}", config["key3"]);
        }
    }
}
```

Run the application to see the configured values:

```
C:\WINDOWS\system32\cmd.exe                              —    □    ×

> dotnet run
Project CustomConfigurationProvider (.NETCoreApp,Version=v1.0) was previously compi
led. Skipping compilation.
key1=value_from_ef_1
key2=value_from_ef_2
key3=value_from_json_3

>
```

Summary

ASP.NET Core provides a very flexible configuration model that supports a number of different file-based options, as well as command-line, in-memory, and environment variables. It works seamlessly with the options model so that you can inject strongly typed settings into your application or framework. You can create your own custom configuration providers as well, which can work with or replace the built-in providers, allowing for extreme flexibility.

1.4.8 Logging

By Steve Smith

ASP.NET Core has built-in support for logging, and allows developers to easily leverage their preferred logging framework's functionality as well. Implementing logging in your application requires a minimal amount of setup code. Once this is in place, logging can be added wherever it is desired.

Sections:

- *Implementing Logging in your Application*
- *Configuring Logging in your Application*
- *Logging Recommendations*
- *Summary*

View or download sample code

Implementing Logging in your Application

Adding logging to a component in your application is done by requesting either an `ILoggerFactory` or an `ILogger<T>` via *Dependency Injection*. If an `ILoggerFactory` is requested, a logger must be created using its `CreateLogger` method. The following example shows how to do this:

```
var logger = loggerFactory.CreateLogger("Catchall Endpoint");
logger.LogInformation("No endpoint found for request {path}", context.Request.Path);
```

When a logger is created, a category name must be provided. The category name specifies the source of the logging events. By convention this string is hierarchical, with categories separated by dot (`.`) characters. Some logging providers have filtering support that leverages this convention, making it easier to locate logging output of interest. In this article's sample application, logging is configured to use the built-in ConsoleLogger (see *Configuring Logging in your Application* below). To see the console logger in action, run the sample application using the `dotnet run` command, and make a request to configured URL (`localhost:5000`). You should see output similar to the following:

You may see more than one log statement per web request you make in your browser, since most browsers will make multiple requests (i.e. for the favicon file) when attempting to load a page. Note that the console logger displayed the log level (`info` in the image above) followed by the category (`[Catchall Endpoint]`), and then the message

---

that was logged.

The call to the log method can utilize a format string with named placeholders (like *{path}*). These placeholders are populated in the order in which they appear by the args values passed into the method call. Some logging providers will store these names along with their mapped values in a dictionary that can later be queried. In the example below, the request path is passed in as a named placeholder:

```
logger.LogInformation("No endpoint found for request {path}", context.Request.Path);
```

In your real world applications, you will want to add logging based on application-level, not framework-level, events. For instance, if you have created a Web API application for managing To-Do Items (see *Building Your First Web API with ASP.NET Core MVC and Visual Studio*), you might add logging around the various operations that can be performed on these items.

The logic for the API is contained within the *TodoController*, which uses *Dependency Injection* to request the services it requires via its constructor. Ideally, classes should follow this example and use their constructor to define their dependencies explicitly as parameters. Rather than requesting an *ILoggerFactory* and creating an instance of *ILogger* explicitly, *TodoController* demonstrates another way to work with loggers in your application - you can request an *ILogger<T>* (where *T* is the class requesting the logger).

```
[Route("api/[controller]")]
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger<TodoController> _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }

    [HttpGet]
    public IEnumerable<TodoItem> GetAll()
    {
        _logger.LogInformation(LoggingEvents.LIST_ITEMS, "Listing all items");
        EnsureItems();
        return _todoRepository.GetAll();
    }
}
```

Within each controller action, logging is done through the use of the local field, *_logger*, as shown on line 17, above. This technique is not limited to controllers, but can be utilized by any of your application services that utilize *Dependency Injection*.

Working with ILogger<T>

As we have just seen, your application can request an instance of `ILogger<T>` as a dependency in a class's constructor, where `T` is the type performing logging. The `TodoController` shows an example of this approach. When this technique is used, the logger will automatically use the type's name as its category name. By requesting an instance of `ILogger<T>`, your class doesn't need to create an instance of a logger via `ILoggerFactory`. You can use this approach anywhere you don't need the additional functionality offered by `ILoggerFactory`.

Logging Verbosity Levels

When adding logging statements to your application, you must specify a LogLevel. The LogLevel allows you to control the verbosity of the logging output from your application, as well as the ability to pipe different kinds of log messages to different loggers. For example, you may wish to log debug messages to a local file, but log errors to the machine's event log or a database.

ASP.NET Core defines six levels of logging verbosity, ordered by increasing importance or severity:

Trace Used for the most detailed log messages, typically only valuable to a developer debugging an issue. These messages may contain sensitive application data and so should not be enabled in a production environment. *Disabled by default.* Example: `Credentials:  {"User":"someuser", "Password":"P@ssword"}`

Debug These messages have short-term usefulness during development. They contain information that may be useful for debugging, but have no long-term value. This is the default most verbose level of logging. Example: `Entering method Configure with flag set to true`

Information These messages are used to track the general flow of the application. These logs should have some long term value, as opposed to `Verbose` level messages, which do not. Example: `Request received for path /foo`

Warning The Warning level should be used for abnormal or unexpected events in the application flow. These may include errors or other conditions that do not cause the application to stop, but which may need to be investigated in the future. Handled exceptions are a common place to use the Warning log level. Examples: `Login failed for IP 127.0.0.1` or `FileNotFoundException for file foo.txt`

Error An error should be logged when the current flow of the application must stop due to some failure, such as an exception that cannot be handled or recovered from. These messages should indicate a failure in the current activity or operation (such as the current HTTP request), not an application-wide failure. Example: `Cannot insert record due to duplicate key violation`

Critical A critical log level should be reserved for unrecoverable application or system crashes, or catastrophic failure that requires immediate attention. Examples: data loss scenarios, out of disk space

The `Logging` package provides helper extension methods for each `LogLevel` value, allowing you to call, for example, `LogInformation`, rather than the more verbose `Log(LogLevel.Information, ...)` method. Each of the `LogLevel`-specific extension methods has several overloads, allowing you to pass in some or all of the following parameters:

string data The message to log.

EventId eventId A numeric id to associate with the log, which can be used to associate a series of logged events with one another. Event IDs should be static and specific to a particular kind of event that is being logged. For instance, you might associate adding an item to a shopping cart as event id 1000 and completing a purchase as event id 1001. This allows intelligent filtering and processing of log statements.

string format A format string for the log message.

object[] args An array of objects to format.

Exception error An exception instance to log.

: The `EventId` type can be implicitly casted to `int`, so you can just pass an `int` to this argument.

: Some loggers, such as the built-in `ConsoleLogger` used in this article, will ignore the `eventId` parameter. If you need to display it, you can include it in the message string. This is done in the following sample so you can easily see the eventId associated with each message, but in practice you would not typically include it in the log message.

In the `TodoController` example, event id constants are defined for each event, and log statements are configured at the appropriate verbosity level based on the success of the operation. In this case, successful operations log as `Information` and not found results are logged as `Warning` (error handling is not shown).

```
[HttpGet]
public IEnumerable<TodoItem> GetAll()
{
    _logger.LogInformation(LoggingEvents.LIST_ITEMS, "Listing all items");
    EnsureItems();
    return _todoRepository.GetAll();
}

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GET_ITEM, "Getting item {0}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GET_ITEM_NOTFOUND, "GetById({0}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

: It is recommended that you perform application logging at the level of your application and its APIs, not at the level of the framework. The framework already has logging built in which can be enabled simply by setting the appropriate logging verbosity level.

To see more detailed logging at the framework level, you can adjust the *LogLevel* specified to your logging provider to something more verbose (like *Debug* or *Trace*). For example, if you modify the *AddConsole* call in the *Configure* method to use *LogLevel.Trace* and run the application, the result shows much more framework-level detail about each request:

The console logger prefixes debug output with "dbug: "; there is no trace level debugging enabled by the framework by default. Each log level has a corresponding four character prefix that is used, so that log messages are consistently aligned.

Log Level	Prefix
Critical	crit
Error	fail
Warning	warn
Information	info
Debug	dbug
Trace	trce

Scopes

In the course of logging information within your application, you can group a set of logical operations within a *scope*. A scope is an `IDisposable` type returned by calling the `ILogger.BeginScope<TState>` method, which lasts from the moment it is created until it is disposed. The built-in `TraceSource` logger returns a scope instance that is responsible for starting and stopping tracing operations. Any logging state, such as a transaction id, is attached to the scope when it is created.

Scopes are not required, and should be used sparingly, if at all. They're best used for operations that have a distinct beginning and end, such as a transaction involving multiple resources.

Configuring Logging in your Application

To configure logging in your ASP.NET Core application, you should resolve `ILoggerFactory` in the `Configure` method of your `Startup` class. ASP.NET Core will automatically provide an instance of `ILoggerFactory` using *Dependency Injection* when you add a parameter of this type to the `Configure` method.

```
    public void Configure(IApplicationBuilder app,
        IHostingEnvironment env,
        ILoggerFactory loggerFactory)
```

Once you've added `ILoggerFactory` as a parameter, you configure loggers within the `Configure` method by calling methods (or extension methods) on the logger factory. We have already seen an example of this configuration at the beginning of this article, when we added console logging by calling `loggerFactory.AddConsole`.

Each logger provides its own set of extension methods to `ILoggerFactory`. The console, debug, and event log loggers allow you to specify the minimum logging level at which those loggers should write log messages. The console and debug loggers provide extension methods accepting a function to filter log messages according to their logging level and/or category (for example, `logLevel => logLevel >= LogLevel.Warning` or `(category, loglevel) => category.Contains("MyController") && loglevel >= LogLevel.Trace`). The event log logger provides a similar overload that takes an `EventLogSettings` instance as argument, which may contain a filtering function in its `Filter` property. The TraceSource logger does not provide any of those overloads, since its logging level and other parameters are based on the `SourceSwitch` and `TraceListener` it uses.

A LoggerFactory instance can optionally be configured with custom `FilterLoggerSettings`. The example below configures custom log levels for different scopes, limiting system and Microsoft built-in logging to warnings while allowing the app to log at debug level by default. The `WithFilter` method returns a new `ILoggerFactory` that will filter the log messages passed to all logger providers registered with it. It does not affect any other `ILoggerFactory` instances, including the original `ILoggerFactory` instance.

```
loggerFactory
    .WithFilter(new FilterLoggerSettings
    {
        { "Microsoft", LogLevel.Warning },
        { "System", LogLevel.Warning },
        { "ToDoApi", LogLevel.Debug }
    })
    .AddConsole();
```

Configuring TraceSource Logging

When running on the full .NET Framework you can configuring logging to use the existing System.Diagnostics.TraceSource libraries and providers, including easy access to the Windows event log. `TraceSource` allows you to route messages to a variety of listeners and is already in use by many organizations.

First, be sure to add the `Microsoft.Extensions.Logging.TraceSource` package to your project (in *project.json*), along with any specific trace source packages you'll be using (in this case, `TextWriterTraceListener`):

```
  "Microsoft.AspNetCore.Mvc": "1.0.0",
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
  "Microsoft.AspNetCore.StaticFiles": "1.0.0",
  "Microsoft.Extensions.Logging": "1.0.0",
  "Microsoft.Extensions.Logging.Console": "1.0.0",
  "Microsoft.Extensions.Logging.Filter": "1.0.0",
  "Microsoft.Extensions.Logging.TraceSource": "1.0.0"
},
"tools": {
  "Microsoft.AspNetCore.Server.IISIntegration.Tools": {
```

The following example demonstrates how to configure a `TraceSourceLogger` instance for an application, logging only `Warning` or higher priority messages. Each call to `AddTraceSource` takes a `TraceListener`. The call

configures a `TextWriterTraceListener` to write to the console window. This log output will be in addition to the console logger that was already added to this sample, but its behavior is slightly different.

```
// add Trace Source logging
var testSwitch = new SourceSwitch("sourceSwitch", "Logging Sample");
testSwitch.Level = SourceLevels.Warning;
loggerFactory.AddTraceSource(testSwitch,
    new TextWriterTraceListener(writer: Console.Out));
```

The `sourceSwitch` is configured to use `SourceLevels.Warning`, so only `Warning` (or higher) log messages are picked up by the `TraceListener` instance.

The API action below logs a warning when the specified `id` is not found:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GET_ITEM, "Getting item {0}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GET_ITEM_NOTFOUND, "GetById({0}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

To test out this code, you can trigger logging a warning by running the app from the console and navigating to `http://localhost:5000/api/Todo/0`. You should see output similar to the following:

The yellow line with the "warn: " prefix, along with the following line, is output by the `ConsoleLogger`. The next line, beginning with "TodoApi.Controllers.TodoController", is output from the TraceSource logger. There are

many other TraceSource listeners available, and the `TextWriterTraceListener` can be configured to use any `TextWriter` instance, making this a very flexible option for logging.

Configuring Other Providers

In addition to the built-in loggers, you can configure logging to use other providers. Add the appropriate package to your *project.json* file, and then configure it just like any other provider. Typically, these packages should include extension methods on `ILoggerFactory` to make it easy to add them.

: The ASP.NET Core team is still working with third party logging providers to publish support for this logging model. Once these ship, we will include links to them here.

You can create your own custom providers as well, to support other logging frameworks or your own internal logging requirements.

Logging Recommendations

The following are some recommendations you may find helpful when implementing logging in your ASP.NET Core applications.

1. Log using the correct `LogLevel`. This will allow you to consume and route logging output appropriately based on the importance of the messages.

2. Log information that will enable errors to be identified quickly. Avoid logging irrelevant or redundant information.

3. Keep log messages concise without sacrificing important information.

4. Although loggers will not log if disabled, consider adding code guards around logging methods to prevent extra method calls and log message setup overhead, especially within loops and performance critical methods.

5. Name your loggers with a distinct prefix so they can easily be filtered or disabled. Remember the `Create<T>` extension will create loggers named with the full name of the class.

6. Use Scopes sparingly, and only for actions with a bounded start and end. For example, the framework provides a scope around MVC actions. Avoid nesting many scopes within one another.

7. Application logging code should be related to the business concerns of the application. Increase the logging verbosity to reveal additional framework-related concerns, rather than implementing yourself.

Summary

ASP.NET Core provides built-in support for logging, which can easily be configured within the `Startup` class and used throughout the application. Logging verbosity can be configured globally and per logging provider to ensure actionable information is logged appropriately. Built-in providers for console and trace source logging are included in the framework; other logging frameworks can easily be configured as well.

1.4.9 File Providers

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.4.10 Dependency Injection

Steve Smith

ASP.NET Core is designed from the ground up to support and leverage dependency injection. ASP.NET Core applications can leverage built-in framework services by having them injected into methods in the Startup class, and application services can be configured for injection as well. The default services container provided by ASP.NET Core provides a minimal feature set and is not intended to replace other containers.

> **Sections:**
>
> - *What is Dependency Injection?*
> - *Using Framework-Provided Services*
> - *Registering Your Own Services*
> - *Service Lifetimes and Registration Options*
> - *Request Services*
> - *Designing Your Services For Dependency Injection*
> - *Replacing the default services container*
> - *Recommendations*
> - *Additional Resources*

View or download sample code

What is Dependency Injection?

Dependency injection (DI) is a technique for achieving loose coupling between objects and their collaborators, or dependencies. Rather than directly instantiating collaborators, or using static references, the objects a class needs in order to perform its actions are provided to the class in some fashion. Most often, classes will declare their dependencies via their constructor, allowing them to follow the Explicit Dependencies Principle. This approach is known as "constructor injection".

When classes are designed with DI in mind, they are more loosely coupled because they do not have direct, hard-coded dependencies on their collaborators. This follows the Dependency Inversion Principle, which states that *"high level modules should not depend on low level modules; both should depend on abstractions."* Instead of referencing specific implementations, classes request abstractions (typically `interfaces`) which are provided to them when they are constructed. Extracting dependencies into interfaces and providing implementations of these interfaces as parameters is also an example of the Strategy design pattern.

When a system is designed to use DI, with many classes requesting their dependencies via their constructor (or properties), it's helpful to have a class dedicated to creating these classes with their associated dependencies. These classes are referred to as *containers*, or more specifically, Inversion of Control (IoC) containers or Dependency Injection (DI) containers. A container is essentially a factory that is responsible for providing instances of types that are requested from it. If a given type has declared that it has dependencies, and the container has been configured to provide the dependency types, it will create the dependencies as part of creating the requested instance. In this way, complex dependency graphs can be provided to classes without the need for any hard-coded object construction. In addition to creating objects with their dependencies, containers typically manage object lifetimes within the application.

ASP.NET Core includes a simple built-in container (represented by the `IServiceProvider` interface) that supports constructor injection by default, and ASP.NET makes certain services available through DI. ASP.NET's container

refs to the types it manages as *services*. Throughout the rest of this article, *services* will refer to types that are managed by ASP.NET Core's IoC container. You configure the built-in container's services in the `ConfigureServices` method in your application's `Startup` class.

: Martin Fowler has written an extensive article on Inversion of Control Containers and the Dependency Injection Pattern. Microsoft Patterns and Practices also has a great description of Dependency Injection.

: This article covers Dependency Injection as it applies to all ASP.NET applications. Dependency Injection within MVC controllers is covered in *Dependency Injection and Controllers*.

Using Framework-Provided Services

The `ConfigureServices` method in the `Startup` class is responsible for defining the services the application will use, including platform features like Entity Framework Core and ASP.NET Core MVC. Initially, the `IServiceCollection` provided to `ConfigureServices` has just a handful of services defined. Below is an example of how to add additional services to the container using a number of extensions methods like `AddDbContext`, `AddIdentity`, and `AddMvc`.

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

The features and middleware provided by ASP.NET, such as MVC, follow a convention of using a single Add*Service* extension method to register all of the services required by that feature.

: You can request certain framework-provided services within `Startup` methods through their parameter lists - see *Application Startup* for more details.

Of course, in addition to configuring the application to take advantage of various framework features, you can also use `ConfigureServices` to configure your own application services.

Registering Your Own Services

You can register your own application services as follows. The first generic type represents the type (typically an interface) that will be requested from the container. The second generic type represents the concrete type that will be instantiated by the container and used to fulfill such requests.

```
services.AddTransient<IEmailSender, AuthMessageSender>();
services.AddTransient<ISmsSender, AuthMessageSender>();
```

: Each `services.Add<service>` calls adds (and potentially configures) services. For example, `services.AddMvc()` adds the services MVC requires.

The `AddTransient` method is used to map abstract types to concrete services that are instantiated separately for every object that requires it. This is known as the service's *lifetime*, and additional lifetime options are described below. It is important to choose an appropriate lifetime for each of the services you register. Should a new instance of the service be provided to each class that requests it? Should one instance be used throughout a given web request? Or should a single instance be used for the lifetime of the application?

In the sample for this article, there is a simple controller that displays character names, called `CharactersController`. Its `Index` method displays the current list of characters that have been stored in the application, and initializes the collection with a handful of characters if none exist. Note that although this application uses Entity Framework Core and the `ApplicationDbContext` class for its persistence, none of that is apparent in the controller. Instead, the specific data access mechanism has been abstracted behind an interface, `ICharacterRepository`, which follows the repository pattern. An instance of `ICharacterRepository` is requested via the constructor and assigned to a private field, which is then used to access characters as necessary.

```csharp
public class CharactersController : Controller
{
    private readonly ICharacterRepository _characterRepository;

    public CharactersController(ICharacterRepository characterRepository)
    {
        _characterRepository = characterRepository;
    }

    // GET: /characters/
    public IActionResult Index()
    {
        PopulateCharactersIfNoneExist();
        var characters = _characterRepository.ListAll();

        return View(characters);
    }

    private void PopulateCharactersIfNoneExist()
    {
        if (!_characterRepository.ListAll().Any())
        {
            _characterRepository.Add(new Character("Darth Maul"));
            _characterRepository.Add(new Character("Darth Vader"));
            _characterRepository.Add(new Character("Yoda"));
            _characterRepository.Add(new Character("Mace Windu"));
        }
    }
}
```

The *ICharacterRepository* simply defines the two methods the controller needs to work with *Character* instances.

```csharp
using System.Collections.Generic;
using DependencyInjectionSample.Models;

namespace DependencyInjectionSample.Interfaces
```

```
{
    public interface ICharacterRepository
    {
        IEnumerable<Character> ListAll();
        void Add(Character character);
    }
}
```

This interface is in turn implemented by a concrete type, `CharacterRepository`, that is used at runtime.

: The way DI is used with the `CharacterRepository` class is a general model you can follow for all of your application services, not just in "repositories" or data access classes.

```
using System.Collections.Generic;
using System.Linq;
using DependencyInjectionSample.Interfaces;

namespace DependencyInjectionSample.Models
{
    public class CharacterRepository : ICharacterRepository
    {
        private readonly ApplicationDbContext _dbContext;

        public CharacterRepository(ApplicationDbContext dbContext)
        {
            _dbContext = dbContext;
        }

        public IEnumerable<Character> ListAll()
        {
            return _dbContext.Characters.AsEnumerable();
        }

        public void Add(Character character)
        {
            _dbContext.Characters.Add(character);
            _dbContext.SaveChanges();
        }
    }
}
```

Note that `CharacterRepository` requests an `ApplicationDbContext` in its constructor. It is not unusual for dependency injection to be used in a chained fashion like this, with each requested dependency in turn requesting its own dependencies. The container is responsible for resolving all of the dependencies in the graph and returning the fully resolved service.

: Creating the requested object, and all of the objects it requires, and all of the objects those require, is sometimes referred to as an *object graph*. Likewise, the collective set of dependencies that must be resolved is typically referred to as a *dependency tree* or *dependency graph*.

In this case, both `ICharacterRepository` and in turn `ApplicationDbContext` must be registered with the services container in `ConfigureServices` in `Startup`. `ApplicationDbContext` is configured with the call to the extension method `AddDbContext<T>`. The following code shows the registration of the `CharacterRepository` type.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseInMemoryDatabase()
    );

    // Add framework services.
    services.AddMvc();

    // Register application services.
    services.AddScoped<ICharacterRepository, CharacterRepository>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));
    services.AddTransient<OperationService, OperationService>();
}
```

Entity Framework contexts should be added to the services container using the `Scoped` lifetime. This is taken care of automatically if you use the helper methods as shown above. Repositories that will make use of Entity Framework should use the same lifetime.

> **:** The main danger to be wary of is resolving a `Scoped` service from a singleton. It's likely in such a case that the service will have incorrect state when processing subsequent requests.

Service Lifetimes and Registration Options

ASP.NET services can be configured with the following lifetimes:

Transient Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services.

Scoped Scoped lifetime services are created once per request.

Singleton Singleton lifetime services are created the first time they are requested (or when `ConfigureServices` is run if you specify an instance there) and then every subsequent request will use the same instance. If your application requires singleton behavior, allowing the services container to manage the service's lifetime is recommended instead of implementing the singleton design pattern and managing your object's lifetime in the class yourself.

Services can be registered with the container in several ways. We have already seen how to register a service implementation with a given type by specifying the concrete type to use. In addition, a factory can be specified, which will then be used to create the instance on demand. The third approach is to directly specify the instance of the type to use, in which case the container will never attempt to create an instance.

To demonstrate the difference between these lifetime and registration options, consider a simple interface that represents one or more tasks as an *operation* with a unique identifier, `OperationId`. Depending on how we configure the lifetime for this service, the container will provide either the same or different instances of the service to the requesting class. To make it clear which lifetime is being requested, we will create one type per lifetime option:

```
using System;

namespace DependencyInjectionSample.Interfaces
{
    public interface IOperation
    {
        Guid OperationId { get; }
```

```
    }

    public interface IOperationTransient : IOperation
    {
    }
    public interface IOperationScoped : IOperation
    {
    }
    public interface IOperationSingleton : IOperation
    {
    }
    public interface IOperationSingletonInstance : IOperation
    {
    }
}
```

We implement these interfaces using a single class, `Operation`, that accepts a `Guid` in its constructor, or uses a new `Guid` if none is provided.

Next, in `ConfigureServices`, each type is added to the container according to its named lifetime:

```
services.AddTransient<IOperationTransient, Operation>();
services.AddScoped<IOperationScoped, Operation>();
services.AddSingleton<IOperationSingleton, Operation>();
services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));
services.AddTransient<OperationService, OperationService>();
```

Note that the `IOperationSingletonInstance` service is using a specific instance with a known ID of `Guid.Empty` so it will be clear when this type is in use. We have also registered an `OperationService` that depends on each of the other `Operation` types, so that it will be clear within a request whether this service is getting the same instance as the controller, or a new one, for each operation type. All this service does is expose its dependencies as properties, so they can be displayed in the view.

```
using DependencyInjectionSample.Interfaces;

namespace DependencyInjectionSample.Services
{
    public class OperationService
    {
        public IOperationTransient TransientOperation { get; }
        public IOperationScoped ScopedOperation { get; }
        public IOperationSingleton SingletonOperation { get; }
        public IOperationSingletonInstance SingletonInstanceOperation { get; }

        public OperationService(IOperationTransient transientOperation,
            IOperationScoped scopedOperation,
            IOperationSingleton singletonOperation,
            IOperationSingletonInstance instanceOperation)
        {
            TransientOperation = transientOperation;
            ScopedOperation = scopedOperation;
            SingletonOperation = singletonOperation;
            SingletonInstanceOperation = instanceOperation;
        }
    }
}
```

To demonstrate the object lifetimes within and between separate individual requests to the application, the sample includes an `OperationsController` that requests each kind of `IOperation` type as well as an

`OperationService`. The `Index` action then displays all of the controller's and service's `OperationId` values.

```
using DependencyInjectionSample.Interfaces;
using DependencyInjectionSample.Services;
using Microsoft.AspNetCore.Mvc;

namespace DependencyInjectionSample.Controllers
{
    public class OperationsController : Controller
    {
        private readonly OperationService _operationService;
        private readonly IOperationTransient _transientOperation;
        private readonly IOperationScoped _scopedOperation;
        private readonly IOperationSingleton _singletonOperation;
        private readonly IOperationSingletonInstance _singletonInstanceOperation;

        public OperationsController(OperationService operationService,
            IOperationTransient transientOperation,
            IOperationScoped scopedOperation,
            IOperationSingleton singletonOperation,
            IOperationSingletonInstance singletonInstanceOperation)
        {
            _operationService = operationService;
            _transientOperation = transientOperation;
            _scopedOperation = scopedOperation;
            _singletonOperation = singletonOperation;
            _singletonInstanceOperation = singletonInstanceOperation;
        }

        public IActionResult Index()
        {
            // viewbag contains controller-requested services
            ViewBag.Transient = _transientOperation;
            ViewBag.Scoped = _scopedOperation;
            ViewBag.Singleton = _singletonOperation;
            ViewBag.SingletonInstance = _singletonInstanceOperation;

            // operation service has its own requested services
            ViewBag.Service = _operationService;
            return View();
        }
    }
}
```

Now two separate requests are made to this controller action:

Observe which of the `OperationId` values varies within a request, and between requests.

- *Transient* objects are always different; a new instance is provided to every controller and every service.

- *Scoped* objects are the same within a request, but different across different requests

- *Singleton* objects are the same for every object and every request (regardless of whether an instance is provided

in `ConfigureServices`)

Request Services

The services available within an ASP.NET request from `HttpContext` are exposed through the `RequestServices` collection.

Request Services represent the services you configure and request as part of your application. When your objects specify dependencies, these are satisfied by the types found in `RequestServices`, not `ApplicationServices`.

Generally, you shouldn't use these properties directly, preferring instead to request the types your classes you require via your class's constructor, and letting the framework inject these dependencies. This yields classes that are easier to test (see *Testing*) and are more loosely coupled.

: Prefer requesting dependencies as constructor parameters to accessing the `RequestServices` collection.

Designing Your Services For Dependency Injection

You should design your services to use dependency injection to get their collaborators. This means avoiding the use of stateful static method calls (which result in a code smell known as static cling) and the direct instantiation of dependent classes within your services. It may help to remember the phrase, New is Glue, when choosing whether to instantiate a type or to request it via dependency injection. By following the SOLID Principles of Object Oriented Design, your classes will naturally tend to be small, well-factored, and easily tested.

What if you find that your classes tend to have way too many dependencies being injected? This is generally a sign that your class is trying to do too much, and is probably violating SRP - the Single Responsibility Principle. See if you can refactor the class by moving some of its responsibilities into a new class. Keep in mind that your `Controller` classes should be focused on UI concerns, so business rules and data access implementation details should be kept in classes appropriate to these separate concerns.

With regard to data access specifically, you can easily inject Entity Framework `DbContext` types into your controllers, assuming you've configured EF in your `Startup` class. However, it is best to avoid depending directly on `DbContext` in your UI project. Instead, depend on an abstraction (like a Repository interface), and restrict knowledge of EF (or any other specific data access technology) to the implementation of this interface. This will reduce the coupling between your application and a particular data access strategy, and will make testing your application code much easier.

Replacing the default services container

The built-in services container is mean to serve the basic needs of the framework and most consumer applications built on it. However, developers who wish to replace the built-in container with their preferred container can easily do so. The `ConfigureServices` method typically returns `void`, but if its signature is changed to return

---

`IServiceProvider`, a different container can be configured and returned. There are many IOC containers available for .NET. In this example, the Autofac package is used.

First, add the appropriate container package(s) to the dependencies property in project.json:

```
"dependencies" : {
  "Autofac": "4.0.0-rc2-237",
  "Autofac.Extensions.DependencyInjection": "4.0.0-rc2-200"
},
```

Next, configure the container in `ConfigureServices` and return an `IServiceProvider`:

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
  // add other framework services

  // Add Autofac
  var containerBuilder = new ContainerBuilder();
  containerBuilder.RegisterModule<DefaultModule>();
  containerBuilder.Populate(services);
  var container = containerBuilder.Build();
  return container.Resolve<IServiceProvider>();
}
```

: When using a third-party DI container, you must change `ConfigureServices` so that it returns `IServiceProvider` instead of `void`.

Finally, configure Autofac as normal in `DefaultModule`:

```
public class DefaultModule : Module
{
  protected override void Load(ContainerBuilder builder)
  {
    builder.RegisterType<CharacterRepository>().As<ICharacterRepository>();
  }
}
```

At runtime, Autofac will be used to resolve types and inject dependencies. Learn more about using Autofac and ASP.NET Core

Recommendations

When working with dependency injection, keep the following recommendations in mind:

- DI is for objects that have complex dependencies. Controllers, services, adapters, and repositories are all examples of objects that might be added to DI.

- Avoid storing data and configuration directly in DI. For example, a user's shopping cart shouldn't typically be added to the services container. Configuration should use the *Options Model*. Similarly, avoid "data holder" objects that only exist to allow access to some other object. It's better to request the actual item needed via DI, if possible.

- Avoid static access to services.

- Avoid service location in your application code.

- Avoid static access to `HttpContext`.

: Like all sets of recommendations, you may encounter situations where ignoring one is required. We have found exceptions to be rare – mostly very special cases within the framework itself.

Remember, dependency injection is an *alternative* to static/global object access patterns. You will not be able to realize the benefits of DI if you mix it with static object access.

Additional Resources

- *Application Startup*
- *Testing*
- Writing Clean Code in ASP.NET Core with Dependency Injection (MSDN)
- Container-Managed Application Design, Prelude: Where does the Container Belong?
- Explicit Dependencies Principle
- Inversion of Control Containers and the Dependency Injection Pattern (Fowler)

1.4.11 Working with Multiple Environments

By Steve Smith

ASP.NET Core introduces improved support for controlling application behavior across multiple environments, such as development, staging, and production. Environment variables are used to indicate which environment the application is running in, allowing the app to be configured appropriately.

> **Sections:**
> - *Development, Staging, Production*
> - *Determining the environment at runtime*
> - *Startup conventions*
> - *Summary*
> - *Additional Resources*

View or download sample code

Development, Staging, Production

ASP.NET Core references a particular environment variable, `ASPNETCORE_ENVIRONMENT` to describe the environment the application is currently running in. This variable can be set to any value you like, but three values are used by convention: `Development`, `Staging`, and `Production`. You will find these values used in the samples and templates provided with ASP.NET Core.

The current environment setting can be detected programmatically from within your application. In addition, you can use the Environment *tag helper* to include certain sections in your *view* based on the current application environment.

: The specified environment name is case insensitive. Whether you set the variable to `Development` or `development` or `DEVELOPMENT` the results will be the same.

Development

This should be the environment used when developing an application. When using Visual Studio, this setting can be specified in your project's debug profiles, such as for IIS Express, shown here:

When you modify the default settings created with the project, your changes are persisted in *launchSettings.json* in the `Properties` folder. This file holds settings specific to each profile Visual Studio is configured to use to launch the application, including any environment variables that should be used. (Debug profiles are discussed in more detail in *Servers*). For example, after adding another profile configured to use IIS Express, but using an `ASPNETCORE_ENVIRONMENT` value of `Staging`, the `launchSettings.json` file in our sample project is shown below:

1.3: launchSettings.json

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:40088/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express (Staging)": {
```

```
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging"
      }
    }
  }
}
```

: Changes made to project profiles or to *launchSettings.json* directly may not take effect until the web server used is restarted (in particular, Kestrel must be restarted before it will detect changes made to its environment).

You can create multiple different launch profiles for various different configurations of your application, including those that require other environment variables.

: Environment variables stored in *launchSettings.json* are not secured in any way and will be part of the source code repository for your project, if you use one. **Never store credentials or other secret data in this file.** If you need a place to store such data, use the *Secret Manager* tool described in *Safe storage of app secrets during development*.

Staging

By convention, a `Staging` environment is a pre-production environment used for final testing before deployment to production. Ideally, its physical characteristics should mirror that of production, so that any issues that may arise in production occur first in the staging environment, where they can be addressed without impact to users.

Production

The `Production` environment is the environment in which the application runs when it is live and being used by end users. This environment should be configured to maximize security, performance, and application robustness. Some common settings that a production environment might have that would differ from development include:

- Turn on caching

- Ensure all client-side resources are bundled, minified, and potentially served from a CDN

- Turn off diagnostic ErrorPages

- Turn on friendly error pages

- Enable production logging and monitoring (for example, Application Insights)

This is by no means meant to be a complete list. It's best to avoid scattering environment checks in many parts of your application. Instead, the recommended approach is to perform such checks within the application's `Startup` class(es) wherever possible

Determining the environment at runtime

The `IHostingEnvironment` service provides the core abstraction for working with environments. This service is provided by the ASP.NET hosting layer, and can be injected into your startup logic via *Dependency Injection*. The ASP.NET Core web site template in Visual Studio uses this approach to load environment-specific configuration files (if present) and to customize the app's error handling settings. In both cases, this behavior is achieved by referring

to the currently specified environment by calling `EnvironmentName` or `IsEnvironment` on the instance of `IHostingEnvironment` passed into the appropriate method.

: If you need to check whether the application is running in a particular environment, use `env.IsEnvironment("environmentname")` since it will correctly ignore case (instead of checking if `env.EnvironmentName == "Development"` for example).

For example, you can use the following code in your Configure method to setup environment specific error handling:

```csharp
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }
    // ...
```

If the app is running in a `Development` environment, then it enables the runtime support necessary to use the "BrowserLink" feature in Visual Studio, development-specific error pages (which typically should not be run in production) and special database error pages (which provide a way to apply migrations and should therefore only be used in development). Otherwise, if the app is not running in a development environment, a standard error handling page is configured to be displayed in response to any unhandled exceptions.

You may need to determine which content to send to the client at runtime, depending on the current environment. For example, in a development environment you generally serve non-minimized scripts and style sheets, which makes debugging easier. Production and test environments should serve the minified versions and generally from a CDN. You can do this using the Environment *tag helper*. The Environment tag helper will only render its contents if the current environment matches one of the environments specified using the `names` attribute.

```html
<environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment names="Staging,Production">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.cs
          asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-v
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

To get started with using tag helpers in your application see *Introduction to Tag Helpers*.

Startup conventions

ASP.NET Core supports a convention-based approach to configuring an application's startup based on the current environment. You can also programmatically control how your application behaves according to which environment it is in, allowing you to create and manage your own conventions.

When an ASP.NET Core application starts, the `Startup` class is used to bootstrap the application, load its configuration settings, etc. (*learn more about ASP.NET startup*). However, if a

---

class exists named `Startup{EnvironmentName}` (for example `StartupDevelopment`), and the `ASPNETCORE_ENVIRONMENT` environment variable matches that name, then that `Startup` class is used instead. Thus, you could configure `Startup` for development, but have a separate `StartupProduction` that would be used when the app is run in production. Or vice versa.

In addition to using an entirely separate `Startup` class based on the current environment, you can also make adjustments to how the application is configured within a `Startup` class. The `Configure()` and `ConfigureServices()` methods support environment-specific versions similar to the `Startup` class itself, of the form `Configure{EnvironmentName}()` and `Configure{EnvironmentName}Services()`. If you define a method `ConfigureDevelopment()` it will be called instead of `Configure()` when the environment is set to development. Likewise, `ConfigureDevelopmentServices()` would be called instead of `ConfigureServices()` in the same environment.

Summary

ASP.NET Core provides a number of features and conventions that allow developers to easily control how their applications behave in different environments. When publishing an application from development to staging to production, environment variables set appropriately for the environment allow for optimization of the application for debugging, testing, or production use, as appropriate.

Additional Resources

- *Configuration*
- *Introduction to Tag Helpers*

1.4.12 Managing Application State

By Steve Smith

In ASP.NET Core, application state can be managed in a variety of ways, depending on when and how the state is to be retrieved. This article provides a brief overview of several options, and focuses on installing and configuring Session state support in ASP.NET Core applications.

Sections

- *Application State Options*
- *Working with HttpContext.Items*
- *Installing and Configuring Session*
- *A Working Sample Using Session*

View or download sample code

Application State Options

Application state refers to any data that is used to represent the current representation of the application. This includes both global and user-specific data. Previous versions of ASP.NET (and even ASP) have had built-in support for global `Application` and `Session` state stores, as well as a variety of other options.

: The `Application` store had the same characteristics as the ASP.NET `Cache`, with fewer capabilities. In ASP.NET Core, `Application` no longer exists; applications written for previous versions of ASP.NET that are migrating to ASP.NET Core replace `Application` with a *Caching* implementation.

Application developers are free to use different state storage providers depending on a variety of factors:

- How long does the data need to persist?

- How large is the data?

- What format is the data?

- Can it be serialized?

- How sensitive was the data? Could it be stored on the client?

Based on answers to these questions, application state in ASP.NET Core apps can be stored or managed in a variety of ways.

HttpContext.Items

The `Items` collection is the best location to store data that is only needed while processing a given request. Its contents are discarded after each request. It is best used as a means of communicating between components or middleware that operate at different points in time during a request, and have no direct relationship with one another through which to pass parameters or return values. See *Working with HttpContext.Items*, below.

Querystring and Post

State from one request can be provided to another request by adding values to the new request's querystring or by POSTing the data. These techniques should not be used with sensitive data, because these techniques require that the data be sent to the client and then sent back to the server. It is also best used with small amounts of data. Querystrings are especially useful for capturing state in a persistent manner, allowing links with embedded state to be created and sent via email or social networks, for use potentially far into the future. However, no assumption can be made about the user making the request, since URLs with querystrings can easily be shared, and care must also be taken to avoid Cross-Site Request Forgery (CSRF) attacks (for instance, even assuming only authenticated users are able to perform actions using querystring-based URLs, an attacker could trick a user into visiting such a URL while already authenticated).

Cookies

Very small pieces of state-related data can be stored in Cookies. These are sent with every request, and so the size should be kept to a minimum. Ideally, only an identifier should be used, with the actual data stored somewhere on the server, keyed to the identifier.

Session

Session storage relies on a cookie-based identifier to access data related to a given browser session (a series of requests from a particular browser and machine). You can't necessarily assume that a session is restricted to a single user, so be careful what kind of information you store in Session. It is a good place to store application state that is specific to a particular session but which doesn't need to be persisted permanently (or which can be reproduced as needed from a persistent store). See *Installing and Configuring Session*, below for more details.

Cache

Caching provides a means of storing and efficiently retrieving arbitrary application data based on developer-defined keys. It provides rules for expiring cached items based on time and other considerations. Learn more about *Caching*.

Configuration

Configuration can be thought of as another form of application state storage, though typically it is read-only while the application is running. Learn more about *Configuration*.

Other Persistence

Any other form of persistent storage, whether using Entity Framework and a database or something like Azure Table Storage, can also be used to store application state, but these fall outside of what ASP.NET supports directly.

Working with HttpContext.Items

The `HttpContext` abstraction provides support for a simple dictionary collection of type `IDictionary<object, object>`, called `Items`. This collection is available from the start of an *HttpRequest'* and is discarded at the end of each request. You can access it by simply assigning a value to a keyed entry, or by requesting the value for a given key.

For example, some simple *Middleware* could add something to the `Items` collection:

```
app.Use(async (context, next) =>
{
    // perform some verification
    context.Items["isVerified"] = true;
    await next.Invoke();
});
```

and later in the pipeline, another piece of middleware could access it:

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Verified request? " + context.Items["isVerified"]);
});
```

: Since keys into `Items` are simple strings, if you are developing middleware that needs to work across many applications, you may wish to prefix your keys with a unique identifier to avoid key collisions (e.g. "MyComponent.isVerified" instead of just "isVerified").

Installing and Configuring Session

ASP.NET Core ships a session package that provides middleware for managing session state. You can install it by including a reference to the `Microsoft.AspNetCore.Session` package in your project.json file.

Once the package is installed, Session must be configured in your application's `Startup` class. Session is built on top of `IDistributedCache`, so you must configure this as well, otherwise you will receive an error.

: If you do not configure at least one `IDistributedCache` implementation, you will get an exception stating "Unable to resolve service for type 'Microsoft.Extensions.Caching.Distributed.IDistributedCache' while attempting to activate 'Microsoft.AspNetCore.Session.DistributedSessionStore'."

ASP.NET ships with several implementations of `IDistributedCache`, including an in-memory option (to be used during development and testing only). To configure session using this in-memory option add the `Microsoft.Extensions.Caching.Memory` package in your project.json file and then add the following to `ConfigureServices`:

```
services.AddDistributedMemoryCache();
services.AddSession();
```

Then, add the following to `Configure` and you're ready to use session in your application code:

```
app.UseSession();
```

You can reference Session from `HttpContext` once it is installed and configured.

: If you attempt to access `Session` before `UseSession` has been called, you will get an `InvalidOperationException` exception stating that "Session has not been configured for this application or request."

: If you attempt to create a new `Session` (i.e. no session cookie has been created yet) after you have already begun writing to the `Response` stream, you will get an `InvalidOperationException` as well, stating that "The session cannot be established after the response has started". This exception may not be displayed in the browser; you may need to view the web server log to discover it, as shown below:

Implementation Details

Session uses a cookie to track and disambiguate between requests from different browsers. By default this cookie is named ".AspNet.Session" and uses a path of "/". Further, by default this cookie does not specify a domain, and is not made available to client-side script on the page (because `CookieHttpOnly` defaults to `true`).

These defaults, as well as the default `IdleTimeout` (used on the server independent from the cookie), can be overridden when configuring `Session` by using `SessionOptions` as shown here:

```
services.AddSession(options =>
{
  options.CookieName = ".AdventureWorks.Session";
  options.IdleTimeout = TimeSpan.FromSeconds(10);
});
```

The `IdleTimeout` is used by the server to determine how long a session can be idle before its contents are abandoned. Each request made to the site that passes through the Session middleware (regardless of whether Session is

read from or written to within that middleware) will reset the timeout. Note that this is independent of the cookie's expiration.

: `Session` is *non-locking*, so if two requests both attempt to modify the contents of session, the last one will win. Further, `Session` is implemented as a *coherent session*, which means that all of the contents are stored together. This means that if two requests are modifying different parts of the session (different keys), they may still impact each other.

ISession

Once session is installed and configured, you refer to it via HttpContext, which exposes a property called `Session` of type `ISession`. You can use this interface to get and set values in `Session`, such as `byte[]`.

```csharp
public interface ISession
{
    bool IsAvailable { get; }
    string Id { get; }
    IEnumerable<string> Keys { get; }
    Task LoadAsync();
    Task CommitAsync();
    bool TryGetValue(string key, out byte[] value);
    void Set(string key, byte[] value);
    void Remove(string key);
    void Clear();
}
```

Because``Session`` is built on top of `IDistributedCache`, you must always serialize the object instances being stored. Thus, the interface works with `byte[]` not simply `object`. However, there are extension methods that make working with simple types such as `String` and `Int32` easier, as well as making it easier to get a byte[] value from session.

```csharp
// session extension usage examples
context.Session.SetInt32("key1", 123);
int? val = context.Session.GetInt32("key1");
context.Session.SetString("key2", "value");
string stringVal = context.Session.GetString("key2");
byte[] result = context.Session.Get("key3");
```

If you're storing more complex objects, you will need to serialize the object to a `byte[]` in order to store them, and then deserialize them from `byte[]` when retrieving them.

A Working Sample Using Session

The associated sample application demonstrates how to work with Session, including storing and retrieving simple types as well as custom objects. In order to see what happens when session expires, the sample has configured sessions to last just 10 seconds:

```csharp
1   {
2       services.AddDistributedMemoryCache();
3
4       services.AddSession(options =>
5       {
6           options.IdleTimeout = TimeSpan.FromSeconds(10);
```

```
7        });
8    }
```

When you first navigate to the web server, it displays a screen indicating that no session has yet been established:

This default behavior is produced by the following middleware in *Startup.cs*, which runs when requests are made that do not already have an established session (note the highlighted sections):

```
1    // main catchall middleware
2    app.Run(async context =>
3    {
4        RequestEntryCollection collection = GetOrCreateEntries(context);
5
6        if (collection.TotalCount() == 0)
7        {
8            await context.Response.WriteAsync("<html><body>");
9            await context.Response.WriteAsync("Your session has not been established.<br>");
10           await context.Response.WriteAsync(DateTime.Now.ToString() + "<br>");
11           await context.Response.WriteAsync("<a href=\"/session\">Establish session</a>.<br>");
12       }
13       else
14       {
15           collection.RecordRequest(context.Request.PathBase + context.Request.Path);
16           SaveEntries(context, collection);
17
18           // Note: it's best to consistently perform all session access before writing anything to res
19           await context.Response.WriteAsync("<html><body>");
20           await context.Response.WriteAsync("Session Established At: " + context.Session.GetString("St
21           foreach (var entry in collection.Entries)
22           {
23               await context.Response.WriteAsync("Request: " + entry.Path + " was requested " + entry.Co
24           }
25
26           await context.Response.WriteAsync("Your session was located, you've visited the site this man
27       }
28       await context.Response.WriteAsync("<a href=\"/untracked\">Visit untracked part of application</a
29       await context.Response.WriteAsync("</body></html>");
30   });
```

`GetOrCreateEntries` is a helper method that will retrieve a `RequestEntryCollection` instance from `Session` if it exists; otherwise, it creates the empty collection and returns that. The collection holds `RequestEntry` instances, which keep track of the different requests the user has made during the current session, and how many requests they've made for each path.

```
1    public class RequestEntry
2    {
```

```
3        public string Path { get; set; }
4        public int Count { get; set; }
5    }
```

```
1    public class RequestEntryCollection
2    {
3        public List<RequestEntry> Entries { get; set; } = new List<RequestEntry>();
4
5        public void RecordRequest(string requestPath)
6        {
7            var existingEntry = Entries.FirstOrDefault(e => e.Path == requestPath);
8            if (existingEntry != null) { existingEntry.Count++; return; }
9
10           var newEntry = new RequestEntry()
11           {
12               Path = requestPath,
13               Count = 1
14           };
15           Entries.Add(newEntry);
16       }
17
18       public int TotalCount()
19       {
20           return Entries.Sum(e => e.Count);
21       }
22   }
```

Fetching the current instance of RequestEntryCollection is done via the GetOrCreateEntries helper method:

```
1    private RequestEntryCollection GetOrCreateEntries(HttpContext context)
2    {
3        RequestEntryCollection collection = null;
4        byte[] requestEntriesBytes = context.Session.Get("RequestEntries");
5
6        if (requestEntriesBytes != null && requestEntriesBytes.Length > 0)
7        {
8            string json = System.Text.Encoding.UTF8.GetString(requestEntriesBytes);
9            return JsonConvert.DeserializeObject<RequestEntryCollection>(json);
10       }
11       if (collection == null)
12       {
13           collection = new RequestEntryCollection();
14       }
15       return collection;
16   }
```

When the entry for the object exists in Session, it is retrieved as a byte[] type, and then deserialized using a MemoryStream and a BinaryFormatter, as shown above. If the object isn't in Session, the method returns a new instance of the RequestEntryCollection.

In the browser, clicking the Establish session hyperlink makes a request to the path "/session", and returns this result:

Refreshing the page results in the count incrementing; returning to the root of the site (after making a few more requests) results in this display, summarizing all of the requests that were made during the current session:

Establishing the session is done in the middleware that handles requests to "/session":

```
1   // establish session
2   app.Map("/session", subApp =>
3   {
4       subApp.Run(async context =>
5       {
6           // uncomment the following line and delete session coookie to generate an error due to sessio
7           // await context.Response.WriteAsync("some content");
8           RequestEntryCollection collection = GetOrCreateEntries(context);
9           collection.RecordRequest(context.Request.PathBase + context.Request.Path);
10          SaveEntries(context, collection);
11          if (context.Session.GetString("StartTime") == null)
12          {
13              context.Session.SetString("StartTime", DateTime.Now.ToString());
14          }
15
16          await context.Response.WriteAsync("<html><body>");
17          await context.Response.WriteAsync($"Counting: You have made {collection.TotalCount()} request
18          await context.Response.WriteAsync("</body></html>");
19      });
20  });
```

Requests to this path will get or create a `RequestEntryCollection`, will add the current path to it, and then will store it in session using the helper method `SaveEntries`, shown below:

```
1   private void SaveEntries(HttpContext context, RequestEntryCollection collection)
2   {
```

```
3        string json = JsonConvert.SerializeObject(collection);
4        byte[] serializedResult = System.Text.Encoding.UTF8.GetBytes(json);
5
6        context.Session.Set("RequestEntries", serializedResult);
7    }
```

`SaveEntries` demonstrates how to serialize a custom object into a `byte[]` for storage in `Session` using a `MemoryStream` and a `BinaryFormatter`.

The sample includes one more piece of middleware worth mentioning, which is mapped to the "/untracked" path. You can see its configuration here:

```
1   // example middleware that does not reference session at all and is configured before app.UseSession
2   app.Map("/untracked", subApp =>
3   {
4       subApp.Run(async context =>
5       {
6           await context.Response.WriteAsync("<html><body>");
7           await context.Response.WriteAsync("Requested at: " + DateTime.Now.ToString("hh:mm:ss.ffff") +
8           await context.Response.WriteAsync("This part of the application isn't referencing Session...
9           await context.Response.WriteAsync("</body></html>");
10      });
11  });
12
13  app.UseSession();
```

Note that this middleware is configured **before** the call to `app.UseSession()` is made (on line 13). Thus, the `Session` feature is not available to this middleware, and requests made to it do not reset the session `IdleTimeout`. You can confirm this behavior in the sample application by refreshing the untracked path several times within 10 seconds, and then return to the application root. You will find that your session has expired, despite no more than 10 seconds having passed between your requests to the application.

1.4.13 Servers

By Steve Smith

ASP.NET Core is completely decoupled from the web server environment that hosts the application. ASP.NET Core supports hosting in IIS and IIS Express, and self-hosting scenarios using the Kestrel and WebListener HTTP servers. Additionally, developers and third party software vendors can create custom servers to host their ASP.NET Core apps.

Sections:

- *Servers and commands*
- *Supported Features by Server*
- *IIS and IIS Express*
- *WebListener*
- *Kestrel*
- *Choosing a server*
- *Custom Servers*
- *Additional Reading*

View or download sample code

Servers and commands

ASP.NET Core was designed to decouple web applications from the underlying HTTP server. Traditionally, ASP.NET apps have been windows-only hosted on Internet Information Server (IIS). The recommended way to run ASP.NET Core applications on Windows is using IIS as a reverse-proxy server. The HttpPlatformHandler module in IIS manages and proxies requests to an HTTP server hosted out-of-process. ASP.NET Core ships with two different HTTP servers:

- Microsoft.AspNetCore.Server.WebListener (AKA WebListener, Windows-only)
- Microsoft.AspNetCore.Server.Kestrel (AKA Kestrel, cross-platform)

ASP.NET Core does not directly listen for requests, but instead relies on the HTTP server implementation to surface the request to the application as a set of *feature interfaces* composed into an HttpContext. While WebListener is Windows-only, Kestrel is designed to run cross-platform. You can configure your application to be hosted by any or all of these servers by specifying commands in your *project.json* file. You can even specify an application entry point for your application, and run it as an executable (using `dotnet run`) rather than hosting it in a separate process.

The default web host for ASP.NET apps developed using Visual Studio is IIS Express functioning as a reverse proxy server for Kestrel. The "Microsoft.AspNetCore.Server.Kestrel" and "Microsoft.AspNetCore.IISPlatformHandler" dependencies are included in *project.json* by default, even with the Empty web site template. Visual Studio provides support for multiple profiles, associated with IIS Express and any other `commands` defined in *project.json*. You can manage these profiles and their settings in the **Debug** tab of your web application project's Properties menu or from the *launchSettings.json* file.

: IIS doesn't support commands. Visual Studio launches IIS Express and loads the application with the selected profile.

The sample project for this article is configured to support each server option in the *project.json* file:

1.4: project.json (truncated)

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

  "dependencies": {
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.Server.WebListener": "1.0.0-rc1-final"
  },

  "commands": {
    "run": "run server.urls=http://localhost:5003",
    "web": "Microsoft.AspNet.Hosting --server Microsoft.AspNet.Server.Kestrel --server.urls http://lo
    "weblistener": "Microsoft.AspNet.Hosting --server WebListener --server.urls http://localhost:5004
  },

  "frameworks": {
    "dnx451": { },
```

The `run` command will launch the application from the `void main` method. The `run` command configures and starts an instance of `Kestrel`.

1.5: program.cs

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNet.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNet.Builder;
using Microsoft.Extensions.Logging;
using Microsoft.AspNet.Server.Kestrel;


namespace ServersDemo
{
    /// <summary>
    /// This demonstrates how the application can be launched in a console application.
    /// Executing the "dnx run" command in the application folder will run this app.
    /// </summary>
    public class Program
    {
        private readonly IServiceProvider _serviceProvider;

        public Program(IServiceProvider serviceProvider)
        {
            _serviceProvider = serviceProvider;
        }

        public Task<int> Main(string[] args)
        {
            //Add command line configuration source to read command line parameters.
            var builder = new ConfigurationBuilder();
            builder.AddCommandLine(args);
            var config = builder.Build();

            using (new WebHostBuilder(config)
```

```
32              .UseServer("Microsoft.AspNet.Server.Kestrel")
33              .Build()
34              .Start())
35          {
36              Console.WriteLine("Started the server..");
37              Console.WriteLine("Press any key to stop the server");
38              Console.ReadLine();
39          }
40          return Task.FromResult(0);
41      }
42   }
43 }
```

Supported Features by Server

ASP.NET defines a number of *Request Features*. The following table lists the WebListener and Kestrel support for request features.

Feature	WebListener	Kestrel
IHttpRequestFeature	Yes	Yes
IHttpResponseFeature	Yes	Yes
IHttpAuthenticationFeature	Yes	No
IHttpUpgradeFeature	Yes (with limits)	Yes
IHttpBufferingFeature	Yes	No
IHttpConnectionFeature	Yes	Yes
IHttpRequestLifetimeFeature	Yes	Yes
IHttpSendFileFeature	Yes	No
IHttpWebSocketFeature	No*	No*
IRequestIdentifierFeature	Yes	No
ITlsConnectionFeature	Yes	Yes
ITlsTokenBindingFeature	Yes	No

Configuration options

You can provide configuration options (by command line parameters or a configuration file) that are read on server startup.

The `Microsoft.AspNetCore.Hosting` command supports server parameters (such as `Kestrel` or `WebListener`) and a `server.urls` configuration key. The `server.urls` configuration key is a semicolon-separated list of URL prefixes that the server should handle.

The *project.json* file shown above demonstrates how to pass the `server.urls` parameter directly:

```
"web": "Microsoft.AspNetCore.Kestrel --server.urls http://localhost:5004"
```

Alternately, a JSON configuration file can be used,

```
"kestrel": "Microsoft.AspNetCore.Hosting"
```

The `hosting.json` can include the settings the server will use (including the server parameter, as well):

```
{
  "server": "Microsoft.AspNetCore.Server.Kestrel",
  "server.urls": "http://localhost:5004/"
}
```

Programmatic configuration

The server hosting the application can be referenced programmatically via the IApplicationBuilder interface, available in the `Configure` method in `Startup`. IApplicationBuilder exposes Server Features of type IFeatureCollection. `IServerAddressesFeature` only expose a `Addresses` property, but different server implementations may expose additional functionality. For instance, WebListener exposes `AuthenticationManager` that can be used to configure the server's authentication:

```
public void Configure(IApplicationBuilder app, IApplicationLifetime lifetime, ILoggerFactory loggerFa
{
    var webListenerInfo = app.ServerFeatures.Get<WebListener>();
    if (webListenerInfo != null)
    {
        webListenerInfo.AuthenticationManager.AuthenticationSchemes =
            AuthenticationSchemes.AllowAnonymous;
    }

    var serverAddress = app.ServerFeatures.Get<IServerAddressesFeature>()?.Addresses.FirstOrDefault()

    app.Run(async (context) =>
    {
        var message = String.Format("Hello World from {0}",
                              serverAddress);
        await context.Response.WriteAsync(message);
    });
}
```

IIS and IIS Express

IIS is the most feature rich server, and includes IIS management functionality and access to other IIS modules. Hosting ASP.NET Core no longer uses the `System.Web` infrastructure used by prior versions of ASP.NET.

HTTPPlatformHandler

In ASP.NET Core on Windows, the web application is hosted by an external process outside of IIS. The HTTP Platform Handler is an IIS 7.5+ module which is responsible for process management of HTTP listeners and used to proxy requests to the processes that it manages.

WebListener

WebListener is a Windows-only HTTP server for ASP.NET Core. It runs directly on the Http.Sys kernel driver, and has very little overhead.

You can add support for WebListener to your ASP.NET application by adding the "Microsoft.AspNetCore.Server.WebListener" dependency in *project.json* and the following command:

```
"web": "Microsoft.AspNetCore.Hosting --server Microsoft.AspNetCore.Server.WebListener --server.urls
```

Kestrel

Kestrel is a cross-platform web server based on libuv, a cross-platform asynchronous I/O library. You add support for Kestrel by including `Microsoft.AspNetCore.Server.Kestrel` in your project's dependencies listed in *project.json*.

Learn more about working with Kestrel to create *Your First ASP.NET Core Application on a Mac Using Visual Studio Code*.

Choosing a server

If you intend to deploy your application on a Windows server, you should run IIS as a reverse proxy server that manages and proxies requests to Kestrel. If deploying on Linux, you should run a comparable reverse proxy server such as Apache or Nginx to proxy requests to Kestrel.

For self-hosting scenarios, such as running in Service Fabric, we recommend using Kestrel without IIS. However, if you require Windows Authentication in a self-hosting scenario, you should choose WebListener.

Custom Servers

You can create your own server in which to host ASP.NET apps, or use other open source servers. When implementing your own server, you're free to implement just the feature interfaces your application needs, though at a minimum you must support IHttpRequestFeature and IHttpResponseFeature.

Since Kestrel is open source, it makes an excellent starting point if you need to implement your own custom server. Like all of ASP.NET Core, you're welcome to contribute any improvements you make back to the project.

Kestrel currently supports a limited number of feature interfaces, but additional features will be added in the future.

Additional Reading

- *Request Features*

1.4.14 Request Features

By Steve Smith

Individual web server features related to how HTTP requests and responses are handled have been factored into separate interfaces. These abstractions are used by individual server implementations and middleware to create and modify the application's hosting pipeline.

Sections:

- *Feature interfaces*
- *Feature collections*
- *Middleware and request features*
- *Summary*
- *Additional Resources*

Feature interfaces

ASP.NET Core defines a number of HTTP feature interfaces in `Microsoft.AspNetCore.Http.Features` which are used by servers to identify the features they support. The following feature interfaces handle requests and return responses:

`IHttpRequestFeature` Defines the structure of an HTTP request, including the protocol, path, query string, headers, and body.

IHttpResponseFeature Defines the structure of an HTTP response, including the status code, headers, and body of the response.

IHttpAuthenticationFeature Defines support for identifying users based on a `ClaimsPrincipal` and specifying an authentication handler.

IHttpUpgradeFeature Defines support for **HTTP Upgrades**, which allow the client to specify which additional protocols it would like to use if the server wishes to switch protocols.

IHttpBufferingFeature Defines methods for disabling buffering of requests and/or responses.

IHttpConnectionFeature Defines properties for local and remote addresses and ports.

IHttpRequestLifetimeFeature Defines support for aborting connections, or detecting if a request has been terminated prematurely, such as by a client disconnect.

IHttpSendFileFeature Defines a method for sending files asynchronously.

IHttpWebSocketFeature Defines an API for supporting web sockets.

IHttpRequestIdentifierFeature Adds a property that can be implemented to uniquely identify requests.

ISessionFeature Defines `ISessionFactory` and `ISession` abstractions for supporting user sessions.

ITlsConnectionFeature Defines an API for retrieving client certificates.

ITlsTokenBindingFeature Defines methods for working with TLS token binding parameters.

: `ISessionFeature` is not a server feature, but is implemented by the `SessionMiddleware` (see *Managing Application State*).

Feature collections

The `Features` property of `HttpContext` provides an interface for getting and setting the available HTTP features for the current request. Since the feature collection is mutable even within the context of a request, middleware can be used to modify the collection and add support for additional features.

Middleware and request features

While servers are responsible for creating the feature collection, middleware can both add to this collection and consume features from the collection. For example, the `StaticFileMiddleware` accesses the `IHttpSendFileFeature` feature. If the feature exists, it is used to send the requested static file from its physical path. Otherwise, a slower alternative method is used to send the file. When available, the `IHttpSendFileFeature` allows the operating system to open the file and perform a direct kernel mode copy to the network card.

Additionally, middleware can add to the feature collection established by the server. Existing features can even be replaced by middleware, allowing the middleware to augment the functionality of the server. Features added to the collection are available immediately to other middleware or the underlying application itself later in the request pipeline.

By combining custom server implementations and specific middleware enhancements, the precise set of features an application requires can be constructed. This allows missing features to be added without requiring a change in server, and ensures only the minimal amount of features are exposed, thus limiting attack surface area and improving performance.

Summary

Feature interfaces define specific HTTP features that a given request may support. Servers define collections of features, and the initial set of features supported by that server, but middleware can be used to enhance these features.

Additional Resources

- *Servers*
- *Middleware*
- *Open Web Interface for .NET (OWIN)*

1.4.15 Open Web Interface for .NET (OWIN)

By Steve Smith and Rick Anderson

ASP.NET Core supports the Open Web Interface for .NET (OWIN). OWIN allows web apps to be decoupled from web servers. It defines a standard way for middleware to be used in a pipeline to handle requests and associated responses. ASP.NET Core applications and middleware can interoperate with OWIN-based applications, servers, and middleware.

Sections:

- *Running OWIN middleware in the ASP.NET pipeline*
- *Using ASP.NET Hosting on an OWIN-based server*
- *Run ASP.NET Core on an OWIN-based server and use its WebSockets support*
- *OWIN keys*
- *Additional Resources*

View or download sample code

Running OWIN middleware in the ASP.NET pipeline

ASP.NET Core's OWIN support is deployed as part of the `Microsoft.AspNetCore.Owin` package. You can import OWIN support into your project by adding this package as a dependency in your *project.json* file:

```
"dependencies": {
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
  "Microsoft.AspNetCore.Owin": "1.0.0"
},
```

OWIN middleware conforms to the OWIN specification, which requires a `Func<IDictionary<string, object>, Task>` interface, and specific keys be set (such as `owin.ResponseBody`). The following simple OWIN middleware displays "Hello World":

```
public Task OwinHello(IDictionary<string, object> environment)
{
    string responseText = "Hello World via OWIN";
    byte[] responseBytes = Encoding.UTF8.GetBytes(responseText);

    // OWIN Environment Keys: http://owin.org/spec/spec/owin-1.0.0.html
    var responseStream = (Stream)environment["owin.ResponseBody"];
```

```
    var responseHeaders = (IDictionary<string, string[]>)environment["owin.ResponseHeaders"];

    responseHeaders["Content-Length"] = new string[] { responseBytes.Length.ToString(CultureInfo.Inva
    responseHeaders["Content-Type"] = new string[] { "text/plain" };

    return responseStream.WriteAsync(responseBytes, 0, responseBytes.Length);
}
```

The sample signature returns a `Task` and accepts an `IDictionary<string, object>` as required by OWIN.

The following code shows how to add the `OwinHello` middleware (shown above) to the ASP.NET pipeline with the `UseOwin` extension method.

```
public void Configure(IApplicationBuilder app)
{
    app.UseOwin(pipeline =>
    {
        pipeline(next => OwinHello);
    });
}
```

You can configure other actions to take place within the OWIN pipeline.

: Response headers should only be modified prior to the first write to the response stream.

: Multiple calls to `UseOwin` is discouraged for performance reasons. OWIN components will operate best if grouped together.

```
app.UseOwin(pipeline =>
{
    pipeline(next =>
    {
        // do something before
        return OwinHello;
        // do something after
    });
});
```

Using ASP.NET Hosting on an OWIN-based server

OWIN-based servers can host ASP.NET applications. One such server is Nowin, a .NET OWIN web server. In the sample for this article, I've included a project that references Nowin and uses it to create an `IServer` capable of self-hosting ASP.NET Core.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting.Server;
using Microsoft.AspNetCore.Hosting.Server.Features;
using Microsoft.AspNetCore.Http.Features;
using Microsoft.AspNetCore.Owin;
using Microsoft.Extensions.Options;
```

```csharp
using Nowin;

namespace NowinSample
{
    public class NowinServer : IServer
    {
        private INowinServer _nowinServer;
        private ServerBuilder _builder;

        public IFeatureCollection Features { get; } = new FeatureCollection();

        public NowinServer(IOptions<ServerBuilder> options)
        {
            Features.Set<IServerAddressesFeature>(new ServerAddressesFeature());
            _builder = options.Value;
        }

        public void Start<TContext>(IHttpApplication<TContext> application)
        {
            // Note that this example does not take into account of Nowin's "server.OnSendingHeaders
            // Ideally we should ensure this method is fired before disposing the context.
            Func<IDictionary<string, object>, Task> appFunc = async env =>
            {
                // The reason for 2 level of wrapping is because the OwinFeatureCollection isn't muta
                // so features can't be added
                var features = new FeatureCollection(new OwinFeatureCollection(env));

                var context = application.CreateContext(features);
                try
                {
                    await application.ProcessRequestAsync(context);
                }
                catch (Exception ex)
                {
                    application.DisposeContext(context, ex);
                    throw;
                }

                application.DisposeContext(context, null);
            };

            // Add the web socket adapter so we can turn OWIN websockets into ASP.NET Core compatible
            // The calling pattern is a bit different
            appFunc = OwinWebSocketAcceptAdapter.AdaptWebSockets(appFunc);

            // Get the server addresses
            var address = Features.Get<IServerAddressesFeature>().Addresses.First();

            var uri = new Uri(address);
            var port = uri.Port;
            IPAddress ip;
            if (!IPAddress.TryParse(uri.Host, out ip))
            {
                ip = IPAddress.Loopback;
            }

            _nowinServer = _builder.SetAddress(ip)
                                   .SetPort(port)
```

```
                                          .SetOwinApp(appFunc)
                                          .Build();
            _nowinServer.Start();
        }

        public void Dispose()
        {
            _nowinServer?.Dispose();
        }
    }
}
```

`IServer` is an interface that requires an `Features` property and a `Start` method.

`Start` is responsible for configuring and starting the server, which in this case is done through a series of fluent API calls that set addresses parsed from the IServerAddressesFeature. Note that the fluent configuration of the `_builder` variable specifies that requests will be handled by the `appFunc` defined earlier in the method. This `Func` is called on each request to process incoming requests.

We'll also add an `IWebHostBuilder` extension to make it easy to add and configure the Nowin server.

```csharp
using System;
using Microsoft.AspNetCore.Hosting.Server;
using Microsoft.Extensions.DependencyInjection;
using Nowin;
using NowinSample;

namespace Microsoft.AspNetCore.Hosting
{
    public static class NowinWebHostBuilderExtensions
    {
        public static IWebHostBuilder UseNowin(this IWebHostBuilder builder)
        {
            return builder.ConfigureServices(services =>
            {
                services.AddSingleton<IServer, NowinServer>();
            });
        }

        public static IWebHostBuilder UseNowin(this IWebHostBuilder builder, Action<ServerBuilder> co
        {
            builder.ConfigureServices(services =>
            {
                services.Configure(configure);
            });
            return builder.UseNowin();
        }
    }
}
```

With this in place, all that's required to run an ASP.NET application using this custom server to call the extension in *Program.cs*:

```csharp
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
```

```
namespace NowinSample
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseNowin()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}
```

Learn more about ASP.NET *Servers*.

Run ASP.NET Core on an OWIN-based server and use its WebSockets support

Another example of how OWIN-based servers' features can be leveraged by ASP.NET Core is access to features like WebSockets. The .NET OWIN web server used in the previous example has support for Web Sockets built in, which can be leveraged by an ASP.NET Core application. The example below shows a simple web app that supports Web Sockets and echoes back everything sent to the server through WebSockets.

```
1    public class Startup
2    {
3        public void Configure(IApplicationBuilder app)
4        {
5            app.Use(async (context, next) =>
6            {
7                if (context.WebSockets.IsWebSocketRequest)
8                {
9                    WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync();
10                   await EchoWebSocket(webSocket);
11               }
12               else
13               {
14                   await next();
15               }
16           });
17
18           app.Run(context =>
19           {
20               return context.Response.WriteAsync("Hello World");
21           });
22       }
23
24       private async Task EchoWebSocket(WebSocket webSocket)
25       {
26           byte[] buffer = new byte[1024];
27           WebSocketReceiveResult received = await webSocket.ReceiveAsync(
28               new ArraySegment<byte>(buffer), CancellationToken.None);
29
30           while (!webSocket.CloseStatus.HasValue)
```

```
31              {
32                  // Echo anything we receive
33                  await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, received.Count),
34                      received.MessageType, received.EndOfMessage, CancellationToken.None);
35
36                  received = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
37                      CancellationToken.None);
38              }
39
40              await webSocket.CloseAsync(webSocket.CloseStatus.Value,
41                  webSocket.CloseStatusDescription, CancellationToken.None);
42          }
43      }
44  }
```

This sample is configured using the same `NowinServer` as the previous one - the only difference is in how the application is configured in its `Configure` method. A test using a simple websocket client demonstrates the application:

OWIN keys

OWIN depends on an `IDictionary<string,object>` object to communicate information throughout an HTTP Request/Response exchange. ASP.NET Core implements the keys listed below. See the primary specification, extensions, and OWIN Key Guidelines and Common Keys.

Request Data (OWIN v1.0.0)

Key	Value (type)	Description
owin.RequestScheme	String	
owin.RequestMethod	String	
owin.RequestPathBase	String	
owin.RequestPath	String	
owin.RequestQueryString	String	
owin.RequestProtocol	String	
owin.RequestHeaders	IDictionary<string,string[]>	
owin.RequestBody	Stream	

Request Data (OWIN v1.1.0)

Key	Value (type)	Description
owin.RequestId	String	Optional

Response Data (OWIN v1.0.0)

Key	Value (type)	Description
owin.ResponseStatusCode	int	Optional
owin.ResponseReasonPhrase	String	Optional
owin.ResponseHeaders	IDictionary<string,string[]>	
owin.ResponseBody	Stream	

Other Data (OWIN v1.0.0)

Key	Value (type)	Description
owin.CallCancelled	CancellationToken	
owin.Version	String	

Common Keys

Key	Value (type)	Description
ssl.ClientCertificate	X509Certificate	
ssl.LoadClientCertAsync	Func<Task>	
server.RemoteIpAddress	String	
server.RemotePort	String	
server.LocalIpAddress	String	
server.LocalPort	String	
server.IsLocal	bool	
server.OnSendingHeaders	Action<Action<object>,object>	

SendFiles v0.3.0

Key	Value (type)	Description
sendfile.SendAsync	See delegate signature	Per Request

Opaque v0.3.0

Key	Value (type)	Description
opaque.Version	`String`	
opaque.Upgrade	`OpaqueUpgrade`	See delegate signature
opaque.Stream	`Stream`	
opaque.CallCancelled	`CancellationToken`	

WebSocket v0.3.0

Key	Value (type)	Description
websocket.Version	`String`	
websocket.Accept	`WebSocketAccept`	See delegate signature.
websocket.AcceptAlt		Non-spec
websocket.SubProtocol	`String`	See RFC6455 Section 4.2.2 Step 5.5
websocket.SendAsync	`WebSocketSendAsync`	See delegate signature.
websocket.ReceiveAsync	`WebSocketReceiveAsync`	See delegate signature.
websocket.CloseAsync	`WebSocketCloseAsync`	See delegate signature.
websocket.CallCancelled	`CancellationToken`	
websocket.ClientCloseStatus	`int`	Optional
websocket.ClientCloseDescription	`String`	Optional

Additional Resources

- *Middleware*

- *Servers*

1.4.16 Choosing the Right .NET For You on the Server

By Daniel Roth

ASP.NET Core is based on the .NET Core project model, which supports building applications that can run cross-platform on Windows, Mac and Linux. When building a .NET Core project you also have a choice of which .NET flavor to target your application at: .NET Framework (CLR), .NET Core (CoreCLR) or Mono. Which .NET flavor should you choose? Let's look at the pros and cons of each one.

.NET Framework

The .NET Framework is the most well known and mature of the three options. The .NET Framework is a mature and fully featured framework that ships with Windows. The .NET Framework ecosystem is well established and has been around for well over a decade. The .NET Framework is production ready today and provides the highest level of compatibility for your existing applications and libraries.

The .NET Framework runs on Windows only. It is also a monolithic component with a large API surface area and a slower release cycle. While the code for the .NET Framework is available for reference it is not an active open source project.

.NET Core

.NET Core is a modular runtime and library implementation that includes a subset of the .NET Framework. .NET Core is supported on Windows, Mac and Linux. .NET Core consists of a set of libraries, called "CoreFX", and a small, optimized runtime, called "CoreCLR". .NET Core is open-source, so you can follow progress on the project and contribute to it on GitHub.

The CoreCLR runtime (Microsoft.CoreCLR) and CoreFX libraries are distributed via NuGet. Because .NET Core has been built as a componentized set of libraries you can limit the API surface area your application uses to just the pieces you need. You can also run .NET Core based applications on much more constrained environments (ex. Windows Server Nano).

The API factoring in .NET Core was updated to enable better componentization. This means that existing libraries built for the .NET Framework generally need to be recompiled to run on .NET Core. The .NET Core ecosystem is relatively new, but it is rapidly growing with the support of popular .NET packages like JSON.NET, AutoFac, xUnit.net and many others.

Developing on .NET Core allows you to target a single consistent platform that can run on multiple platforms.

Mono

Mono is a port of the .NET Framework built primarily for non-Windows platforms. Mono is open source and cross-platform. It also shares a similar API factoring to the .NET Framework, so many existing managed libraries work on Mono today. Mono is a good proving ground for cross-platform development while cross-platform support in .NET Core matures.

Summary

The .NET Core project model makes .NET development available for more scenarios than ever before. With .NET Core you have the option to target your application at existing available .NET platforms. Which .NET flavor you pick will depend on your specific scenarios, timelines, feature requirements and compatibility requirements.

1.5 MVC

1.5.1 Overview of ASP.NET MVC

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.5.2 Models

Model Binding

By Rachel Appel

Sections:

Introduction to model binding

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

How model binding works

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

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

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

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

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

MVC will try to bind request data to the action parameters by name. MVC will look for values for each parameter using the parameter name and the names of its public settable properties. In the above example, the only action parameter is named `id`, which MVC binds to the value with the same name in the route values. In addition to route values MVC will bind data from various parts of the request and it does so in a set order. Below is a list of the data sources in the order that model binding looks through them:

1. `Form values`: These are form values that go in the HTTP request using the POST method. (including jQuery POST requests).

2. `Route values`: The set of route values provided by routing.

3. `Query strings`: The query string part of the URI.

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

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

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

In order for binding to happen the class must have a public default constructor and member to be bound must be public writable properties. When model binding happens the class will only be instantiated using the public default constructor, then the properties can be set.

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

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

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

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

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

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

Customize model binding behavior with attributes

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

- `[BindRequired]`: This attribute adds a model state error if binding cannot occur.
- `[BindNever]`: Tells the model binder to never bind to this parameter.
- `[FromHeader], [FromQuery], [FromRoute], [FromForm]`: Use these to specify the exact binding source you want to apply.
- `[FromServices]`: This attribute uses *dependency injection* to bind parameters from services.
- `[FromBody]`: Use the configured formatters to bind data from the request body. The formatter is selected based on content type of the request.
- `[ModelBinder]`: Used to override the default model binder, binding source and name.

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

Binding formatted data from the request body

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

: The `JsonInputFormatter` is the default formatter and it is based off of Json.NET.

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

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

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

Model Validation

By Rachel Appel

In this article:

Sections

- *Introduction to model validation*
- *Validation Attributes*
- *Model State*
- *Handling Model State Errors*
- *Manual validation*
- *Custom validation*
- *Client side validation*
- *IClientModelValidator*
- *Remote validation*

Introduction to model validation

Before an app stores data in a database, the app must validate the data. Data must be checked for potential security threats, verified that it is appropriately formatted by type and size, and it must conform to your rules. Validation is necessary although it can be redundant and tedious to implement. In MVC, validation happens on both the client and server.

Fortunately, .NET has abstracted validation into validation attributes. These attributes contain validation code, thereby reducing the amount of code you must write.

Validation Attributes

Validation attributes are a way to configure model validation so it's similar conceptually to validation on fields in database tables. This includes constraints such as assigning data types or required fields. Other types of validation include applying patterns to data to enforce business rules, such as a credit card, phone number, or email address. Validation attributes make enforcing these requirements much simpler and easier to use.

Below is an annotated `Movie` model from an app that stores information about movies and TV shows. Most of the properties are required and several string properties have length requirements. Additionally, there is a numeric range restriction in place for the `Price` property from 0 to $999.99, along with a custom validation attribute.

```
public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [Required]
    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; }

    [Required]
    [Range(0, 999.99)]
    public decimal Price { get; set; }

    [Required]
    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

Simply reading through the model reveals the rules about data for this app, making it easier to maintain the code. Below are several popular built-in validation attributes:

- `[CreditCard]`: Validates the property has a credit card format.

- `[Compare]`: Validates two properties in a model match.

- `[EmailAddress]`: Validates the property has an email format.

- `[Phone]`: Validates the property has a telephone format.

- `[Range]`: Validates the property value falls within the given range.

- `[RegularExpression]`: Validates that the data matches the specified regular expression.

- `[Required]`: Makes a property required.

- `[StringLength]`: Validates that a string property has at most the given maximum length.

- `[Url]`: Validates the property has a URL format.

MVC supports any attribute that derives from `ValidationAttribute` for validation purposes. Many useful validation attributes can be found in the System.ComponentModel.DataAnnotations namespace.

There may be instances where you need more features than built-in attributes provide. For those times, you can create custom validation attributes by deriving from `ValidationAttribute` or changing your model to implement `IValidatableObject`.

Model State

Model state represents validation errors in submitted HTML form values.

MVC will continue validating fields until reaches the maximum number of errors (200 by default). You can configure this number by inserting the following code into the `ConfigureServices` method in the `Startup.cs` file:

```
services.AddMvc(options => options.MaxModelValidationErrors = 50);
```

Handling Model State Errors

Model validation occurs prior to each controller action being invoked, and it is the action method's responsibility to inspect ModelState.IsValid and react appropriately. In many cases, the appropriate reaction is to return some kind of error response, ideally detailing the reason why model validation failed.

Some apps will choose to follow a standard convention for dealing with model validation errors, in which case a filter may be an appropriate place to implement such a policy. You should test how your actions behave with valid and invalid model states.

Manual validation

After model binding and validation are complete, you may want to repeat parts of it. For example, a user may have entered text in a field expecting an integer, or you may need to compute a value for a model's property.

You may need to run validation manually. To do so, call the `TryValidateModel` method, as shown here:

```
TryValidateModel(movie);
```

Custom validation

Validation attributes work for most validation needs. However, some validation rules are specific to your business, as they're not just generic data validation such as ensuring a field is required or that it conforms to a range of values. For these scenarios, custom validation attributes are a great solution. Creating your own custom validation attributes in MVC is easy. Just inherit from the `ValidationAttribute`, and override the `IsValid` method. The `IsValid` method accepts two parameters, the first is an object named *value* and the second is a `ValidationContext` object named *validationContext*. *Value* refers to the actual value from the field that your custom validator is validating.

In the following sample, a business rule states that users may not set the genre to *Classic* for a movie released after 1960. The `[ClassicMovie]` attribute checks the genre first, and if it is a classic, then it checks the release date to see that it is later than 1960. If it is released after 1960, validation fails. The attribute accepts an integer parameter representing the year that you can use to validate data. You can capture the value of the parameter in the attribute's constructor, as shown here:

```
public class ClassicMovieAttribute : ValidationAttribute, IClientModelValidator
{
    private int _year;
```

```
    public ClassicMovieAttribute(int Year)
    {
        _year = Year;
    }


    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
    {
        Movie movie = (Movie)validationContext.ObjectInstance;

        if (movie.Genre == Genre.Classic && movie.ReleaseDate.Year > _year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }
```

The `movie` variable above represents a `Movie` object that contains the data from the form submission to validate. In this case, the validation code checks the date and genre in the `IsValid` method of the `ClassicMovieAttribute` class as per the rules. Upon successful validation `IsValid` returns a `ValidationResult.Success` code, and when validation fails, a `ValidationResult` with an error message. When a user modifies the `Genre` field and submits the form, the `IsValid` method of the `ClassicMovieAttribute` will verify whether the movie is a classic. Like any built-in attribute, apply the `ClassicMovieAttribute` to a property such as `ReleaseDate` to ensure validation happens, as shown in the previous code sample. Since the example works only with `Movie` types, a better option is to use `IValidatableObject` as shown in the following paragraph.

Alternatively, this same code could be placed in the model by implementing the `Validate` method on the `IValidatableObject` interface. While custom validation attributes work well for validating individual properties, implementing `IValidatableObject` can be used to implement class-level validation as seen here.

```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
    if (Genre == Genre.Classic && ReleaseDate.Year > _classicYear)
    {
        yield return new ValidationResult(
            "Classic movies must have a release year earlier than " + _classicYear,
            new[] { "ReleaseDate" });
    }
}
```

Client side validation

Client side validation is a great convenience for users. It saves time they would otherwise spend waiting for a round trip to the server. In business terms, even a few fractions of seconds multiplied hundreds of times each day adds up to be a lot of time, expense, and frustration. Straightforward and immediate validation enables users to work more efficiently and produce better quality input and output.

You must have a view with the proper JavaScript script references in place for client side validation to work as you see here.

```
<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.3.min.js"></script>
```

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/jquery.validate.min.js"></script>
<script src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.6/jquery.validate.unob
```

MVC uses validation attributes in addition to type metadata from model properties to validate data and display any

error messages using JavaScript. When you use MVC to render form elements from a model using Tag Helpers or HTML helpers it will add HTML 5 data- attributes in the form elements that need validation, as shown below. MVC generates the `data-` attributes for both built-in and custom attributes. You can display validation errors on the client using the relevant tag helpers as shown here:

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

The tag helpers above render the HTML below. Notice that the `data-` attributes in the HTML output correspond to the validation attributes for the `ReleaseDate` property. The `data-val-required` attribute below contains an error message to display if the user doesn't fill in the release date field, and that message displays in the accompanying `<span>` element.

```
<form action="/movies/Create" method="post">
  <div class="form-horizontal">
    <h4>Movie</h4>
    <div class="text-danger"></div>
    <div class="form-group">
      <label class="col-md-2 control-label" for="ReleaseDate">ReleaseDate</label>
      <div class="col-md-10">
        <input class="form-control" type="datetime"
        data-val="true" data-val-required="The ReleaseDate field is required."
        id="ReleaseDate" name="ReleaseDate" value="" />
        <span class="text-danger field-validation-valid"
        data-valmsg-for="ReleaseDate" data-valmsg-replace="true"></span>
      </div>
    </div>
    </div>
</form>
```

Client-side validation prevents submission until the form is valid. The Submit button runs JavaScript that either submits the form or displays error messages.

MVC determines type attribute values based on the .NET data type of a property, possibly overridden using `[DataType]` attributes. The base `[DataType]` attribute does no real server-side validation. Browsers choose their own error messages and display those errors however they wish, however the jQuery Validation Unobtrusive package can override the messages and display them consistently with others. This happens most obviously when users apply `[DataType]` subclasses such as `[EmailAddress]`.

IClientModelValidator

You may create client side logic for your custom attribute, and unobtrusive validation will execute it on the client for you automatically as part of validation. The first step is to control what data- attributes are added by implementing the `IClientModelValidator` interface as shown here:

```
public void AddValidation(ClientModelValidationContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }
```

```
    MergeAttribute(context.Attributes, "data-val", "true");
    MergeAttribute(context.Attributes, "data-val-classicmovie", GetErrorMessage());

    var year = _year.ToString(CultureInfo.InvariantCulture);
    MergeAttribute(context.Attributes, "data-val-classicmovie-year", year);
}
```

Attributes that implement this interface can add HTML attributes to generated fields. Examining the output for the `ReleaseDate` element reveals HTML that is similar to the previous example, except now there is a `data-val-classicmovie` attribute that was defined in the `AddValidation` method of `IClientModelValidator`.

```
<input class="form-control" type="datetime"
data-val="true"
data-val-classicmovie="Classic movies must have a release year earlier than 1960"
data-val-classicmovie-year="1960"
data-val-required="The ReleaseDate field is required."
id="ReleaseDate" name="ReleaseDate" value="" />
```

Unobtrusive validation uses the data in the `data-` attributes to display error messages. However, jQuery doesn't know about rules or messages until you add them to jQuery's `validator` object. This is shown in the example below that adds a method named `classicmovie` containing custom client validation code to the jQuery `validator` object.

```
$(function () {
    jQuery.validator.addMethod('classicmovie',
        function (value, element, params) {
            // Get element value. Classic genre has value '0'.
            var genre = $(params[0]).val(),
                year = params[1],
                date = new Date(value);
            if (genre && genre.length > 0 && genre[0] === '0') {
                // Since this is a classic movie, invalid if release date is after given year.
                return date.getFullYear() <= year;
            }

            return true;
        });

    jQuery.validator.unobtrusive.adapters.add('classicmovie',
        [ 'element', 'year' ],
        function (options) {
            var element = $(options.form).find('select#Genre')[0];
            options.rules['classicmovie'] = [element, parseInt(options.params['year'])];
            options.messages['classicmovie'] = options.message;
        });
}(jQuery));
```

Now jQuery has the information to execute the custom JavaScript validation as well as the error message to display if that validation code returns false.

Remote validation

Remote validation is a great feature to use when you need to validate data on the client against data on the server. For example, your app may need to verify whether an email or user name is already in use, and it must query a large amount of data to do so. Downloading large sets of data for validating one or a few fields consumes too many resources. It may also expose sensitive information. An alternative is to make a round-trip request to validate a field.

You can implement remote validation in a two step process. First, you must annotate your model with the `[Remote]` attribute. The `[Remote]` attribute accepts multiple overloads you can use to direct client side JavaScript to the appropriate code to call. The example points to the `VerifyEmail` action method of the `Users` controller.

```
public class User
{
    [Remote(action: "VerifyEmail", controller: "Users")]
    public string Email { get; set; }
}
```

The second step is putting the validation code in the corresponding action method as defined in the `[Remote]` attribute. It returns a `JsonResult` that the client side can use to proceed or pause and display an error if needed.

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyEmail(string email)
{
    if (!_userRepository.VerifyEmail(email))
    {
        return Json(data: $"Email {email} is already in use.");
    }

    return Json(data: true);
}
```

Now when users enter an email, JavaScript in the view makes a remote call to see if that email has been taken, and if so, then displays the error message. Otherwise, the user can submit the form as usual.

Formatting Response Data

By Steve Smith

ASP.NET Core MVC has built-in support for formatting response data, using fixed formats or in response to client specifications.

Sections

- *Format-Specific Action Results*
- *Content Negotiation*
- *Configuring Formatters*
- *Response Format URL Mappings*

View or download sample from GitHub.

Format-Specific Action Results

Some action result types are specific to a particular format, such as `JsonResult` and `ContentResult`. Actions can return specific results that are always formatted in a particular manner. For example, returning a `JsonResult` will return JSON-formatted data, regardless of client preferences. Likewise, returning a `ContentResult` will return plain-text-formatted string data (as will simply returning a string).

: An action isn't required to return any particular type; MVC supports any object return value. If an action returns an `IActionResult` implementation and the controller inherits from `Controller`, developers have many helper

---

methods corresponding to many of the choices. Results from actions that return objects that are not `IActionResult` types will be serialized using the appropriate `IOutputFormatter` implementation.

To return data in a specific format from a controller that inherits from the `Controller` base class, use the built-in helper method `Json` to return JSON and `Content` for plain text. Your action method should return either the specific result type (for instance, `JsonResult`) or `IActionResult`.

Returning JSON-formatted data:

```
// GET: api/authors
[HttpGet]
public JsonResult Get()
{
    return Json(_authorRepository.List());
}
```

Sample response from this action:

Note that the content type of the response is `application/json`, shown both in the list of network requests and in the Response Headers section. Also note the list of options presented by the browser (in this case, Microsoft Edge) in the Accept header in the Request Headers section. The current technique is ignoring this header; obeying it is discussed below.

To return plain text formatted data, use `ContentResult` and the `Content` helper:

```
// GET api/authors/about
[HttpGet("About")]
public ContentResult About()
{
    return Content("An API listing authors of docs.asp.net.");
}
```

A response from this action:

Note in this case the `Content-Type` returned is `text/plain`. You can also achieve this same behavior using just a string response type:

```
// GET api/authors/version
[HttpGet("version")]
public string Version()
{
    return "Version 1.0.0";
}
```

: For non-trivial actions with multiple return types or options (for example, different HTTP status codes based on the result of operations performed), prefer `IActionResult` as the return type.

Content Negotiation

Content negotiation (*conneg* for short) occurs when the client specifies an Accept header. The default format used by ASP.NET Core MVC is JSON. Content negotiation is implemented by `ObjectResult`. It is also built into the status code specific action results returned from the helper methods (which are all based on `ObjectResult`). You can also return a model type (a class you've defined as your data transfer type) and the framework will automatically wrap it in an `ObjectResult` for you.

The following action method uses the `Ok` and `NotFound` helper methods:

```
// GET: api/authors/search?namelike=th
[HttpGet("Search")]
public IActionResult Search(string namelike)
```

```
{
    var result = _authorRepository.GetByNameSubstring(namelike);
    if (!result.Any())
    {
        return NotFound(namelike);
    }
    return Ok(result);
}
```

A JSON-formatted response will be returned unless another format was requested and the server can return the requested format. You can use a tool like Fiddler to create a request that includes an Accept header and specify another format. In that case, if the server has a *formatter* that can produce a response in the requested format, the result will be returned in the client-preferred format.

In the above screenshot, the Fiddler Composer has been used to generate a request, specifying `Accept: application/xml`. By default, ASP.NET Core MVC only supports JSON, so even when another format is specified, the result returned is still JSON-formatted. You'll see how to add additional formatters in the next section.

Controller actions can return POCOs (Plain Old CLR Objects), in which case ASP.NET MVC will automatically create an `ObjectResult` for you that wraps the object. The client will get the formatted serialized object (JSON format is the default; you can configure XML or other formats). If the object being returned is `null`, then the framework will return a `204 No Content` response.

Returning an object type:

```
// GET api/authors/ardalis
[HttpGet("{alias}")]
public Author Get(string alias)
{
    return _authorRepository.GetByAlias(alias);
}
```

In the sample, a request for a valid author alias will receive a 200 OK response with the author's data. A request for an invalid alias will receive a 204 No Content response. Screenshots showing the response in XML and JSON formats are shown below.

Content Negotiation Process Content *negotiation* only takes place if an `Accept` header appears in the request. When a request contains an accept header, the framework will enumerate the media types in the accept header in preference order and will try to find a formatter that can produce a response in one of the formats specified by the accept header. In case no formatter is found that can satisfy the client's request, the framework will try to find the first formatter that can produce a response (unless the developer has configured the option on `MvcOptions` to return 406 Not Acceptable instead). If the request specifies XML, but the XML formatter has not been configured, then the JSON formatter will be used. More generally, if no formatter is configured that can provide the requested format, then the first formatter than can format the object is used. If no header is given, the first formatter that can handle the object to

be returned will be used to serialize the response. In this case, there isn't any negotiation taking place - the server is determining what format it will use.

: If the Accept header contains /, the Header will be ignored unless `RespectBrowserAcceptHeader` is set to true on `MvcOptions`.

Browsers and Content Negotiation   Unlike typical API clients, web browsers tend to supply `Accept` headers that include a wide array of formats, including wildcards. By default, when the framework detects that the request is coming from a browser, it will ignore the `Accept` header and instead return the content in the application's configured default format (JSON unless otherwise configured). This provides a more consistent experience when using different browsers to consume APIs.

If you would prefer your application honor browser accept headers, you can configure this as part of MVC's configuration by setting `RespectBrowserAcceptHeader` to `true` in the `ConfigureServices` method in *Startup.cs*.

```
services.AddMvc(options =>
{
  options.RespectBrowserAcceptHeader = true; // false by default
}
```

Configuring Formatters

If your application needs to support additional formats beyond the default of JSON, you can add these as additional dependencies in *project.json* and configure MVC to support them. There are separate formatters for input and output. Input formatters are used by *Model Binding*; output formatters are used to format responses. You can also configure *Custom Formatters*.

Adding   XML   Format   Support   To   add   support   for   XML   formatting,   add   the   "Microsoft.AspNetCore.Mvc.Formatters.Xml" package to your *project.json*'s list of dependencies.

Add the XmlSerializerFormatters to MVC's configuration in *Startup.cs*:

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

    services.AddScoped<IAuthorRepository, AuthorRepository>();
}
```

Alternately, you can add just the output formatter:

```
services.AddMvc(options =>
{
  options.OutputFormatters.Add(new XmlSerializerOutputFormatter());
});
```

These two approaches will serialize results using System.Xml.Serialization.XmlSerializer. If you prefer, you can use the System.Runtime.Serialization.DataContractSerializer by adding its associated formatter:

```
services.AddMvc(options =>
{
  options.OutputFormatters.Add(new XmlDataContractSerializerOutputFormatter());
});
```

Once you've added support for XML formatting, your controller methods should return the appropriate format based on the request's `Accept` header, as this Fiddler example demonstrates:

You can see in the Inspectors tab that the Raw GET request was made with an `Accept:  application/xml` header set. The response pane shows the `Content-Type:  application/xml` header, and the `Author` object has been serialized to XML.

Use the Composer tab to modify the request to specify `application/json` in the `Accept` header. Execute the request, and the response will be formatted as JSON:

In this screenshot, you can see the request sets a header of `Accept:  application/json` and the response specifies the same as its `Content-Type`. The `Author` object is shown in the body of the response, in JSON format.

Forcing a Particular Format   If you would like to restrict the response formats for a specific action you can, you can apply the `[Produces]` filter. The `[Produces]` filter specifies the response formats for a specific action (or controller). Like most *Filters*, this can be applied at the action, controller, or global scope.

```
[Produces("application/json")]
public class AuthorsController
```

The `[Produces]` filter will force all actions within the `AuthorsController` to return JSON-formatted responses, even if other formatters were configured for the application and the client provided an `Accept` header requesting a different, available format. See *Filters* to learn more, including how to apply filters globally.

Special Case Formatters   Some special cases are implemented using built-in formatters. By default, `string` return types will be formatted as *text/plain* (*text/html* if requested via `Accept` header). This behavior can be removed by removing the `TextOutputFormatter`. You remove formatters in the `Configure` method in *Startup.cs* (shown below). Actions that have a model object return type will return a 204 No Content response when returning `null`. This behavior can be removed by removing the `HttpNoContentOutputFormatter`. The following code removes the `TextOutputFormatter` and *HttpNoContentOutputFormatter'*.

```
services.AddMvc(options =>
{
  options.OutputFormatters.RemoveType<TextOutputFormatter>();
  options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();
});
```

Without the `TextOutputFormatter`, `string` return types return 406 Not Acceptable, for example. Note that if an XML formatter exists, it will format `string` return types if the `TextOutputFormatter` is removed.

Without the `HttpNoContentOutputFormatter`, null objects are formatted using the configured formatter. For example, the JSON formatter will simply return a response with a body of `null`, while the XML formatter will return an empty XML element with the attribute `xsi:nil="true"` set.

Response Format URL Mappings

Clients can request a particular format as part of the URL, such as in the query string or part of the path, or by using a format-specific file extension such as .xml or .json. The mapping from request path should be specified in the route the API is using. For example:

```
[FormatFilter]
public class ProductsController
{
  [Route("[controller]/[action]/{id}.{format?}")]
  public Product GetById(int id)
```

This route would allow the requested format to be specified as an optional file extension. The `[FormatFilter]` attribute checks for the existence of the format value in the `RouteData` and will map the response format to the appropriate formatter when the response is created.

<div align="center">1.5: Examples</div>

Route	Formatter
`/products/GetById/5`	The default output formatter
`/products/GetById/5.json`	The JSON formatter (if configured)
`/products/GetById/5.xml`	The XML formatter (if configured)

Custom Formatters

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.5.3 Views

Views Overview

By Steve Smith

ASP.NET MVC Core controllers can return formatted results using *views*.

Sections

- *What are Views?*
- *Benefits of Using Views*
- *Creating a View*
- *How do Controllers Specify Views?*
- *Passing Data to Views*
- *More View Features*

What are Views?

In the Model-View-Controller (MVC) pattern, the *view* encapsulates the presentation details of the user's interaction with the app. Views are HTML templates with embedded code that generate content to send to the client. Views use *Razor syntax*, which allows code to interact with HTML with minimal code or ceremony.

ASP.NET Core MVC views are *.cshtml* files stored by default in a *Views* folder within the application. Typically, each controller will have its own folder, in which are views for specific controller actions.

In addition to action-specific views, *partial views*, *layouts, and other special view files* can be used to help reduce repetition and allow for reuse within the app's views.

Benefits of Using Views

Views provide separation of concerns within an MVC app, encapsulating user interface level markup separately from business logic. ASP.NET MVC views use *Razor syntax* to make switching between HTML markup and server side logic painless. Common, repetitive aspects of the app's user interface can easily be reused between views using *layout and shared directives* or *partial views*.

Creating a View

Views that are specific to a controller are created in the *Views/[ControllerName]* folder. Views that are shared among controllers are placed in the */Views/Shared* folder. Name the view file the same as its associated controller action, and add the *.cshtml* file extension. For example, to create a view for the *About* action on the *Home* controller, you would create the *About.cshtml* file in the */Views/Home* folder.

A sample view file (*About.cshtml*):

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

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

Razor code is denoted by the @ symbol. C# statements are run within Razor code blocks set off by curly braces ({ }), such as the assignment of "About" to the `ViewData["Title"]` element shown above. Razor can be used to display values within HTML by simply referencing the value with the @ symbol, as shown within the `<h2>` and `<h3>` elements above.

This view focuses on just the portion of the output for which it is responsible. The rest of the page's layout, and other common aspects of the view, are specified elsewhere. Learn more about *layout and shared view logic*.

How do Controllers Specify Views?

Views are typically returned from actions as a `ViewResult`. Your action method can create and return a `ViewResult` directly, but more commonly if your controller inherits from `Controller`, you'll simply use the `View` helper method, as this example demonstrates:

HomeController.cs

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

The `View` helper method has several overloads to make returning views easier for app developers. You can optionally specify a view to return, as well as a model object to pass to the view.

When this action returns, the *About.cshtml* view shown above is rendered:

View Discovery   When an action returns a view, a process called *view discovery* takes place. This process determines which view file will be used. Unless a specific view file is specified, the runtime looks for a controller-specific view first, then looks for matching view name in the *Shared* folder.

When an action returns the `View` method, like so `return View();`, the action name is used as the view name. For example, if this were called from an action method named "Index", it would be equivalent to passing in a view name of "Index". A view name can be explicitly passed to the method (`return View("SomeView");`). In both of these cases, view discovery searches for a matching view file in:

1. Views/<ControllerName>/<ViewName>.cshtml

2. Views/Shared/<ViewName>.cshtml

: We recommend following the convention of simply returning `View()` from actions when possible, as it results in more flexible, easier to refactor code.

A view file path can be provided, instead of a view name. In this case, the *.cshtml* extension must be specified as part of the file path. The path should be relative to the application root (and can optionally start with "/" or "~/"). For example: `return View("Views/Home/About.cshtml");`

: *Partial views* and *view components* use similar (but not identical) discovery mechanisms.

: You can customize the default convention regarding where views are located within the app by using a custom `IViewLocationExpander`.

: View names may be case sensitive depending on the underlying file system. For compatibility across operating systems, always match case between controller and action names and associated view folders and filenames.

Passing Data to Views

You can pass data to views using several mechanisms. The most robust approach is to specify a *model* type in the view (commonly referred to as a *viewmodel*, to distinguish it from business domain model types), and then pass an instance of this type to the view from the action. We recommend you use a model or view model to pass data to a view. This allows the view to take advantage of strong type checking. You can specify a model for a view using the `@model` directive:

```
@model WebApplication1.ViewModels.Address
<h2>Contact</h2>
<address>
    @Model.Street<br />
    @Model.City, @Model.State @Model.PostalCode<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>
```

Once a model has been specified for a view, the instance sent to the view can be accessed in a strongly-typed manner using `@Model` as shown above. To provide an instance of the model type to the view, the controller passes it as a parameter:

```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    var viewModel = new Address()
    {
        Name = "Microsoft",
        Street = "One Microsoft Way",
        City = "Redmond",
        State = "WA",
        PostalCode = "98052-6399"
    };
    return View(viewModel);
}
```

There are no restrictions on the types that can be provided to a view as a model. We recommend passing Plain Old CLR Object (POCO) view models with little or no behavior, so that business logic can be encapsulated elsewhere in the app. An example of this approach is the *Address* viewmodel used in the example above:

```
namespace WebApplication1.ViewModels
{
    public class Address
    {
        public string Name { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
    }
}
```

: Nothing prevents you from using the same classes as your business model types and your display model types. However, keeping them separate allows your views to vary independently from your domain or persistence model, and can offer some security benefits as well (for models that users will send to the app using *model binding*).

Loosely Typed Data In addition to strongly typed views, all views have access to a loosely typed collection of data. This same collection can be referenced through either the `ViewData` or `ViewBag` properties on controllers and views. The `ViewBag` property is a wrapper around `ViewData` that provides a dynamic view over that collection. It is not a separate collection.

`ViewData` is a dictionary object accessed through `string` keys. You can store and retrieve objects in it, and you'll need to cast them to a specific type when you extract them. You can use `ViewData` to pass data from a controller to views, as well as within views (and partial views and layouts). String data can be stored and used directly, without the need for a cast.

Set some values for `ViewData` in an action:

```
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello";
    ViewData["Address"]  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

Work with the data in a view:

```
@{
    // Requires cast
    var address = ViewData["Address"] as Address;
}

@ViewData["Greeting"] World!

<address>
    @address.Name<br />
    @address.Street<br />
```

```
    @address.City, @address.State @address.PostalCode
</address>
```

The `ViewBag` objects provides dynamic access to the objects stored in `ViewData`. This can be more convenient to work with, since it doesn't require casting. The same example as above, using `ViewBag` instead of a strongly typed `address` instance in the view:

```
@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br />
    @ViewBag.Address.Street<br />
    @ViewBag.Address.City, @ViewBag.Address.State @ViewBag.Address.PostalCode
</address>
```

: Since both refer to the same underlying `ViewData` collection, you can mix and match between `ViewData` and `ViewBag` when reading and writing values, if convenient.

Dynamic Views   Views that do not declare a model type but have a model instance passed to them can reference this instance dynamically. For example, if an instance of `Address` is passed to a view that doesn't declare an `@model`, the view would still be able to refer to the instance's properties dynamically as shown:

```
<address>
    @Model.Street<br />
    @Model.City, @Model.State @Model.PostalCode<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>
```

This feature can offer some flexibility, but does not offer any compilation protection or IntelliSense. If the property doesn't exist, the page will fail at runtime.

More View Features

Tag helpers make it easy to add server-side behavior to existing HTML tags, avoiding the need to use custom code or helpers within views. Tag helpers are applied as attributes to HTML elements, which are ignored by editors that aren't familiar with them, allowing view markup to be edited and rendered in a variety of tools. Tag helpers have many uses, and in particular can make *working with forms* much easier.

Generating custom HTML markup can be achieved with many built-in *HTML Helpers*, and more complex UI logic (potentially with its own data requirements) can be encapsulated in *View Components*. View components provide the same separation of concerns that controllers and views offer, and can eliminate the need for actions and views to deal with data used by common UI elements.

Like many other aspects of ASP.NET Core, views support *dependency injection*, allowing services to be *injected into views*.

Razor Syntax

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

Layout

By Steve Smith

Views frequently share visual and programmatic elements. In this article, you'll learn how to use common layouts, share directives, and run common code before rendering views in your ASP.NET app.

Sections

- *What is a Layout*
- *Specifying a Layout*
- *Importing Shared Directives*
- *Running Code Before Each View*

What is a Layout

Most web apps have a common layout that provides the user with a consistent experience as they navigate from page to page. The layout typically includes common user interface elements such as the app header, navigation or menu elements, and footer.

Common HTML structures such as scripts and stylesheets are also frequently used by many pages within an app. All of these shared elements may be defined in a *layout* file, which can then be referenced by any view used within the app. Layouts reduce duplicate code in views, helping them follow the Don't Repeat Yourself (DRY) principle.

By convention, the default layout for an ASP.NET app is named `_Layout.cshtml`. The Visual Studio ASP.NET Core MVC project template includes this layout file in the `Views/Shared` folder:

```
▲ 🔒📂 Views
    ▷ 🔒📁 Account
    ▷ 🔒📁 Home
    ▷ 🔒📁 Manage
    ▲ 🔒📂 Shared
          🔒📄 _Layout.cshtml
          🔒📄 _LoginPartial.cshtml
          🔒📄 _ValidationScriptsPartial.cshtml
          🔒📄 Error.cshtml
      🔒📄 _ViewImports.cshtml
      🔒📄 _ViewStart.cshtml
```

This layout defines a top level template for views in the app. Apps do not require a layout, and apps can define more than one layout, with different views specifying different layouts.

An example `_Layout.cshtml`:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebApplication1</title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.m
               asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
               asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-te
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navb
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">WebAppli
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
                </ul>
                @await Html.PartialAsync("_LoginPartial")
```
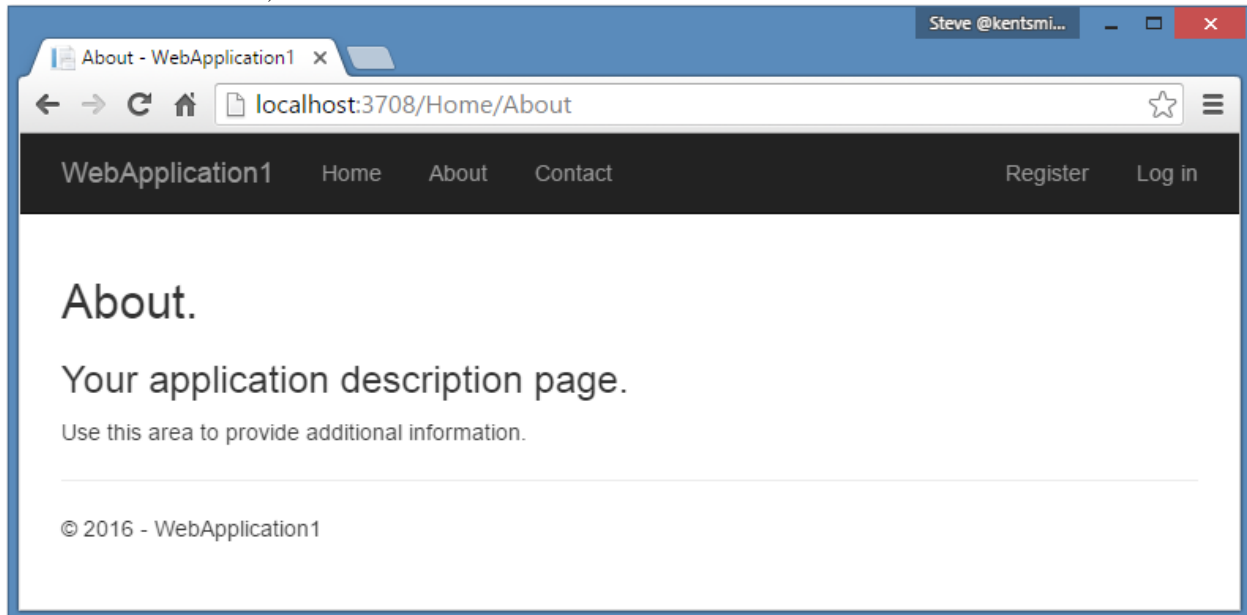
```
                </div>
            </div>
        </div>
        <div class="container body-content">
            @RenderBody()
            <hr />
            <footer>
                <p>&copy; 2016 - WebApplication1</p>
            </footer>
        </div>

        <environment names="Development">
            <script src="~/lib/jquery/dist/jquery.js"></script>
            <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
            <script src="~/js/site.js" asp-append-version="true"></script>
        </environment>
        <environment names="Staging,Production">
            <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
                    asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                    asp-fallback-test="window.jQuery">
            </script>
            <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/bootstrap.min.js"
                    asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                    asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
            </script>
            <script src="~/js/site.min.js" asp-append-version="true"></script>
        </environment>

        @RenderSection("scripts", required: false)
</body>
</html>
```

Specifying a Layout

Razor views have a `Layout` property. Individual views specify a layout by setting this property:

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

The layout specified can use a full path (example: `/Views/Shared/_Layout.cshtml`) or a partial name (example: `_Layout`). When a partial name is provided, the Razor view engine will search for the layout file using its standard discovery process. The controller-associated folder is searched first, followed by the `Shared` folder. This discovery process is identical to the one used to discover *partial views*.

By default, every layout must call `RenderBody`. Wherever the call to `RenderBody` is placed, the contents of the view will be rendered.

Sections   A layout can optionally reference one or more *sections*, by calling `RenderSection`. Sections provide a way to organize where certain page elements should be placed. Each call to `RenderSection` can specify whether that section is required or optional. If a required section is not found, an exception will be thrown. Individual views specify the content to be rendered within a section using the `@section` Razor syntax. If a view defines a section, it must be rendered (or an error will occur).

An example `@section` definition in a view:

```
@section Scripts {
  <script type="text/javascript" src="/scripts/main.js"></script>
}
```

In the code above, validation scripts are added to the `scripts` section on a view that includes a form. Other views in the same application might not require any additional scripts, and so wouldn't need to define a scripts section.

Sections defined in a view are available only in its immediate layout page. They cannot be referenced from partials, view components, or other parts of the view system.

Ignoring sections By default, the body and all sections in a content page must all be rendered by the layout page. The Razor view engine enforces this by tracking whether the body and each section have been rendered.

To instruct the view engine to ignore the body or sections, call the `IgnoreBody` and `IgnoreSection` methods.

The body and every section in a Razor page must be either rendered or ignored.

Importing Shared Directives

Views can use Razor directives to do many things, such as importing namespaces or performing *dependency injection*. Directives shared by many views may be specified in a common `_ViewImports.cshtml` file. The `_ViewImports` file supports the following directives:

- `@addTagHelper`

- `@removeTagHelper`

- `@tagHelperPrefix`

- `@using`

- `@model`

- `@inherits`

- `@inject`

The file does not support other Razor features, such as functions and section definitions.

A sample `_ViewImports.cshtml` file:

```
@using WebApplication1
@using WebApplication1.Models
@using WebApplication1.Models.AccountViewModels
@using WebApplication1.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The `_ViewImports.cshtml` file for an ASP.NET Core MVC app is typically placed in the `Views` folder. A `_ViewImports.cshtml` file can be placed within any folder, in which case it will only be applied to views within that folder and its subfolders. `_ViewImports` files are processed starting at the root level, and then for each folder leading up to the location of the view itself, so settings specified at the root level may be overridden at the folder level.

For example, if a root level `_ViewImports.cshtml` file specifies `@model` and `@addTagHelper`, and another `_ViewImports.cshtml` file in the controller-associated folder of the view specifies a different `@model` and adds another `@addTagHelper`, the view will have access to both tag helpers and will use the latter `@model`.

If multiple `_ViewImports.cshtml` files are run for a view, combined behavior of the directives included in the `ViewImports.cshtml` files will be as follows:

- `@addTagHelper`, `@removeTagHelper`: all run, in order

- `@tagHelperPrefix`: the closest one to the view overrides any others

- `@model`: the closest one to the view overrides any others

- `@inherits`: the closest one to the view overrides any others

- `@using`: all are included; duplicates are ignored

- `@inject`: for each property, the closest one to the view overrides any others with the same property name

Running Code Before Each View

If you have code you need to run before every view, this should be placed in the `_ViewStart.cshtml` file. By convention, the `_ViewStart.cshtml` file is located in the `Views` folder. The statements listed in `_ViewStart.cshtml` are run before every full view (not layouts, and not partial views). Like *ViewImports.cshtml*, `_ViewStart.cshtml` is hierarchical. If a `_ViewStart.cshtml` file is defined in the controller-associated view folder, it will be run after the one defined in the root of the `Views` folder (if any).

A sample `_ViewStart.cshtml` file:

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

The file above specifies that all views will use the `_Layout.cshtml` layout.

: Neither `_ViewStart.cshtml` nor `_ViewImports.cshtml` are typically placed in the `/Views/Shared` folder. The app-level versions of these files should be placed directly in the `/Views` folder.

Dynamic vs Strongly Typed Views

: We are currently working on this topic.

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

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the issue.
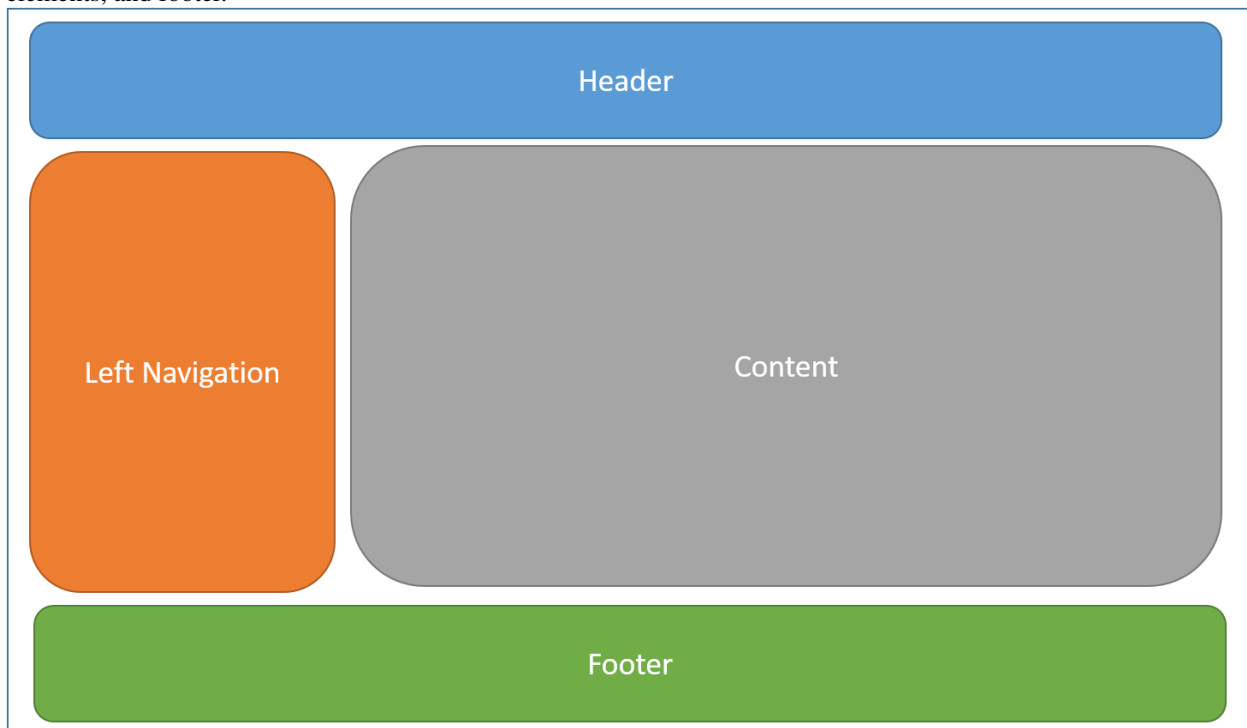
Learn more about how you can contribute on GitHub.

Learn more about Dynamic vs Strongly Typed Views.

Working with Forms

By Rick Anderson, Dave Paquette and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes *Tag Helpers* and how they can help you productively create robust HTML forms. We recommend you read *Introduction to Tag Helpers* before you read this document.

In many cases, *HTML Helpers* provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers do not replace HTML Helpers and there is not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it is mentioned.

Sections:

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route

- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)

- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.

- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
  <!-- Input and Submit elements -->
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
 </form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is very difficult, the Form Tag Helper provides this service for you.

Using a named route   The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a *route* named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
  asp-route-returnurl="@ViewData["ReturnUrl"]"
  method="post" class="form-horizontal" role="form">
```

Note With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>" />
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`, that is the attribute value literally is part of an expression. The name of the expression is what's used for the `asp-for` attribute value.

- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property

- Will not overwrite the HTML `type` attribute value when one is specified

- Generates HTML5 validation attributes from data annotation attributes applied to model properties

- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
 'RegisterViewModel' does not contain a definition for 'Email' and no
 extension method 'Email' accepting a first argument of type 'RegisterViewModel'
 could be found (are you missing a using directive or an assembly reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

```
  <form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
           data-val-email="The Email Address field is not a valid e-mail address."
           data-val-required="The Email Address field is required."
           id="Email" name="Email" value="" /> <br>
    Password:
    <input type="password" data-val="true"
           data-val-required="The Password field is required."
           id="Password" name="Password" /><br>
    <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see *Model Validation*). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it is displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional

details about the rule, for example, `data-val-maxlength-max="1024"`.

HTML Helper alternatives to Input Tag Helper `Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` will not. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper does not. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

Expression names The `asp-for` attribute value is a [ModelExpression](#) and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
    var joe = "Joe";
}
<input asp-for="@joe" />
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe" />
```

Navigating child properties You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

Expression names and Collections   Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
```

The following Razor shows how to iterate over a collection:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }
```

```
    </table>
    <button type="submit">Save</button>
</form>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
        <label asp-for="@Model[i].Name"></label>
        @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
        <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

> **Note** Always use `for` (and *not* `foreach`) to iterate over a list. Evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **Note** The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The Textarea Tag Helper tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a `<textarea>` element.

- Provides strong typing.

- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

public class DescriptionViewModel
{
    [MinLength(5)]
    [MaxLength(1024)]
    public string Description { get; set; }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
```

```
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name

- HTML Helper alternative: `Html.LabelFor`.

The Label Tag Helper provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code

- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;
public class SimpleViewModel
{
    [Required]
    [EmailAddress]
    [Display(Name = "Email Address")]
    public string Email { get; set; }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the <label> element:

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

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the <input> element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The Validation Message Tag Helper (which displays a validation message for a single property on your model), and the Validation Summary Tag Helper (which displays a summary of validation errors). The Input Tag Helper adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.

- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.

- HTML Helper alternative: `Html.ValidationMessageFor`

The Validation Message Tag Helper is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email">
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true">
```

You generally use the Validation Message Tag Helper after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> **Note** You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See *Model Validation* for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
          data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute

- HTML Helper alternative: `@Html.ValidationSummary`

The Validation Summary Tag Helper is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

asp-validation-summary	Validation messages displayed
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample   In the following example, the data model is decorated with `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```csharp
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```html
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ValidationSummary.ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

```html
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid e-mail address."
   data-val="true"> <br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.

- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The Select Tag Helper `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

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

public class CountryViewModel
{
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; } = new List<SelectListItem>
    {
        new SelectListItem { Value = "MX", Text = "Mexico" },
        new SelectListItem { Value = "CA", Text = "Canada" },
        new SelectListItem { Value = "US", Text = "USA"  },
    };
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```csharp
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```csharp
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country +  " selected";
        return RedirectToAction("IndexSuccess", new { message = msg});
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

```html
<form method="post" action="/">
  <select id="Country" name="Country">
```

```html
    <option value="MX">Mexico</option>
    <option selected="selected" value="CA">Canada</option>
    <option value="US">USA</option>
  </select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

> **Note** We do not recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```html
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding   It's often convenient to use `<select>` with an `enum` property and generate the SelectListItem elements from the `enum` values.

Sample:

```csharp
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

```csharp
using System.ComponentModel.DataAnnotations;

public enum CountryEnum
{
    Mexico,
    USA,
    Canada,
    France,
    Germany,
    Spain
}
```

The GetEnumSelectList method generates a SelectList object for an enum.

```html
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()"> >
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can decorate your enumerator list with the `Display` attribute to get a richer UI:

```csharp
using System.ComponentModel.DataAnnotations;

public enum CountryEnum
{
    [Display(Name = "United Mexican States")]
    Mexico,
    [Display(Name = "United States of America")]
    USA,
```

```
    Canada,
    France,
    Germany,
    Spain
}
```

The following HTML is generated:

```html
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

Option Group   The HTML <optgroup> element is generated when the view model contains one or more SelectList-Group objects.

The CountryViewModelGroup groups the SelectListItem elements into the "North America" and "Europe" groups:

```csharp
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
```

```
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
        };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
}
```

The two groups are shown below:
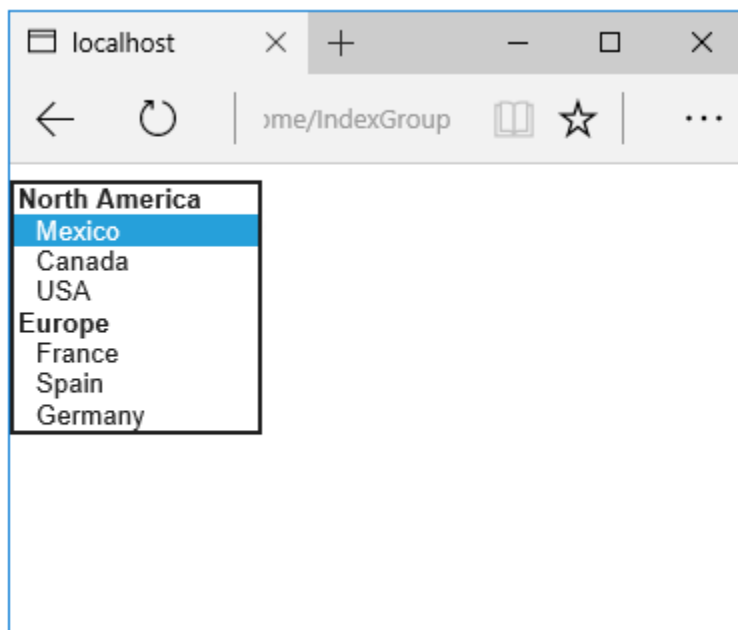
The generated HTML:

```html
<form method="post" action="/Home/IndexGroup">
    <select id="Country" name="Country">
        <optgroup label="North America">
            <option value="MEX">Mexico</option>
            <option value="CAN">Canada</option>
            <option value="US">USA</option>
        </optgroup>
        <optgroup label="Europe">
```

```
                <option value="FR">France</option>
                <option value="ES">Spain</option>
                <option value="DE">Germany</option>
            </optgroup>
        </select>
        <br /><button type="submit">Register</button>
        <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
    </form>
```

Multiple select   The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

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

public class CountryViewModelIEnumerable
{
    public IEnumerable<string> CountryCodes { get; set; }

    public List<SelectListItem> Countries { get; } = new List<SelectListItem>
    {
        new SelectListItem { Value = "MX", Text = "Mexico" },
        new SelectListItem { Value = "CA", Text = "Canada" },
        new SelectListItem { Value = "US", Text = "USA"     },
        new SelectListItem { Value = "FR", Text = "France" },
        new SelectListItem { Value = "ES", Text = "Spain"  },
        new SelectListItem { Value = "DE", Text = "Germany"}
    };
}
```

With the following view:

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```html
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

No selection   To allow for no selection, add a "not specified" option to the select list. If the property is a value type, you'll have to make it nullable.

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country" asp-items="Model.Countries">
        <option value="">&lt;none&gt;</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements is not limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct <option> element will be selected (contain the selected="selected" attribute) depending on the current Country value.

```
<form method="post" action="/Home/IndexEmpty">
    <select id="Country" name="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA" selected="selected">Canada</option>
        <option value="US">USA</option>
    </select>
```

```
        <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

Additional Resources

- *Tag Helpers*

- HTML Form element

- Request Verification Token

- *Model Binding*

- *Model Validation*

- data annotations

- Code snippets for this document.

HTML Helpers

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

Tag Helpers

Introduction to Tag Helpers

By Rick Anderson

- *What are Tag Helpers?*

- *What Tag Helpers provide*

- *Managing Tag Helper scope*

- *IntelliSense support for Tag Helpers*

- *Tag Helpers compared to HTML Helpers*

- *Tag Helpers compared to Web Server Controls*

- *Customizing the Tag Helper element font*

- *Additional Resources*

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

What Tag Helpers provide

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

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

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

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

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

Generates the following HTML:

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

The asp-for attribute is made available by the For property in the LabelTagHelper. See *Authoring Tag Helpers* for more information.

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

@addTagHelper makes Tag Helpers available   If you create a new ASP.NET Core web app named *Authoring-TagHelpers* (with no authentication), the following *Views/_ViewImports.cshtml* file will be added to your project:

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

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

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

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

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

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

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

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

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

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

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

Opting out of individual elements You can disable a Tag Helper at the element level with the Tag Helper opt-out character ("!"). For example, `Email` validation is disabled in the `<span>` with the Tag Helper opt-out character:

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

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

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

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

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

IntelliSense support for Tag Helpers   When you create a new ASP.NET web app in Visual Studio, it adds "Microsoft.AspNetCore.Razor.Tools" to the *project.json* file. This is the package that adds Tag Helper tooling.

Consider writing an HTML `<label>` element. As soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:
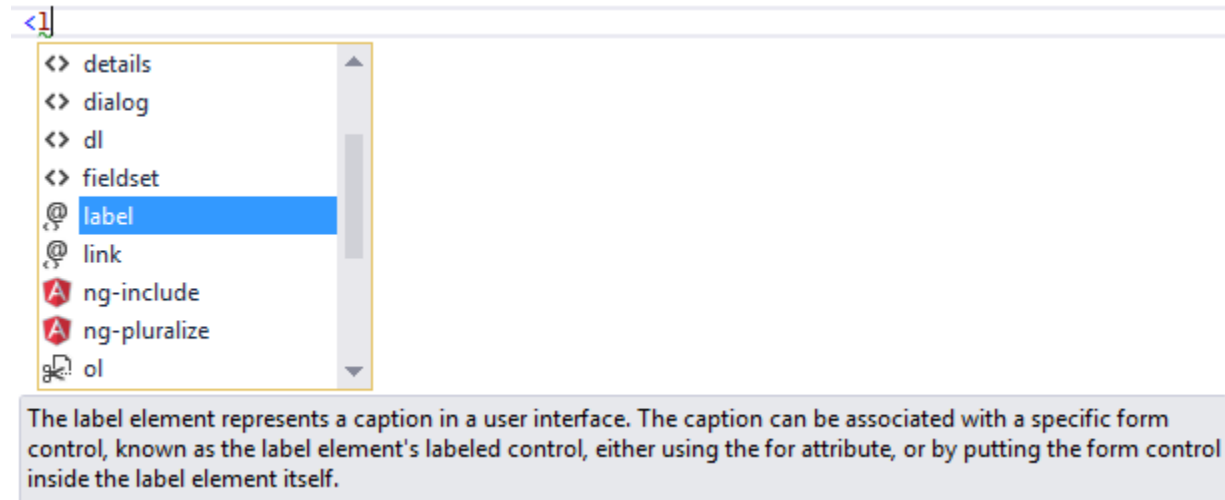
Not only do you get HTML help, but the icon (the "@" symbol with "<>" under it).

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

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

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

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

```
<label |
         accesskey
      aria-...
   @  asp-for
      class
      contenteditable
      contextmenu
      data-...
      db
      dir
```
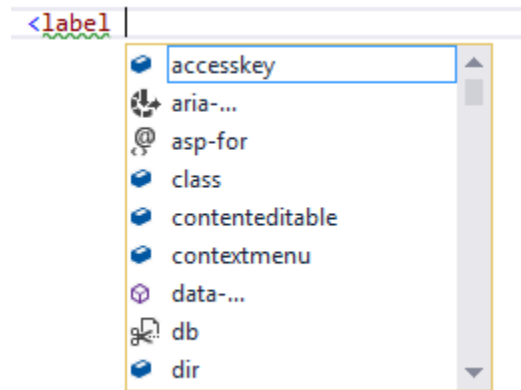
IntelliSense statement completion allows you to enter the tab key to complete the statement with the selected value:

```
<label as|
      @  asp-for
         class
```
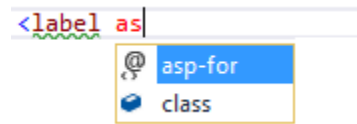
As soon as a Tag Helper attribute is entered, the tag and attribute fonts change. Using the default Visual Studio "Blue" or "Light" color theme, the font is bold purple. If you're using the "Dark" theme the font is bold teal. The images in this document were taken using the default theme.

```
<label asp-for
```

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

```
                    <label asp-for="e"
                              ConfirmPassword
  string RegisterViewModel.Email { get; set; }   Email
                              Equals
                              GetHashCode
                              GetType
                              Password
                              ToString
```
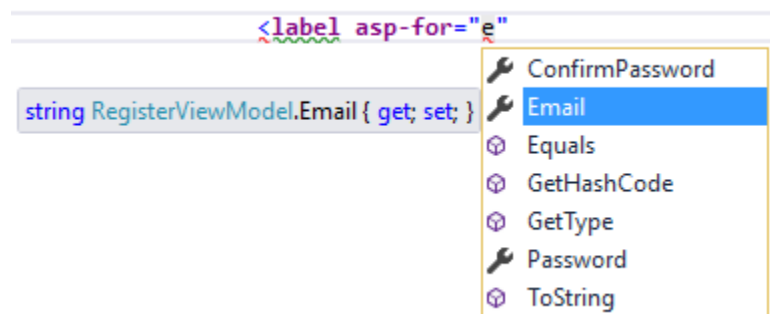
IntelliSense lists the properties and methods available to the model on the page. The rich IntelliSense environment helps you select the CSS class:

```
<input asp-for="Email" cl|
```

⬢	class
⚡	onclick
⚡	onclose
⚡	ondblclick

```
<input asp-for="Email" class="co|"
```

⬢	body-content
⬢	bs-example-control-sizing
⬢	bs-glyphicons
B	carousel-control
B	col
⬢	col-1
⬢	col-2
⬢	col-3
⬢	col-4
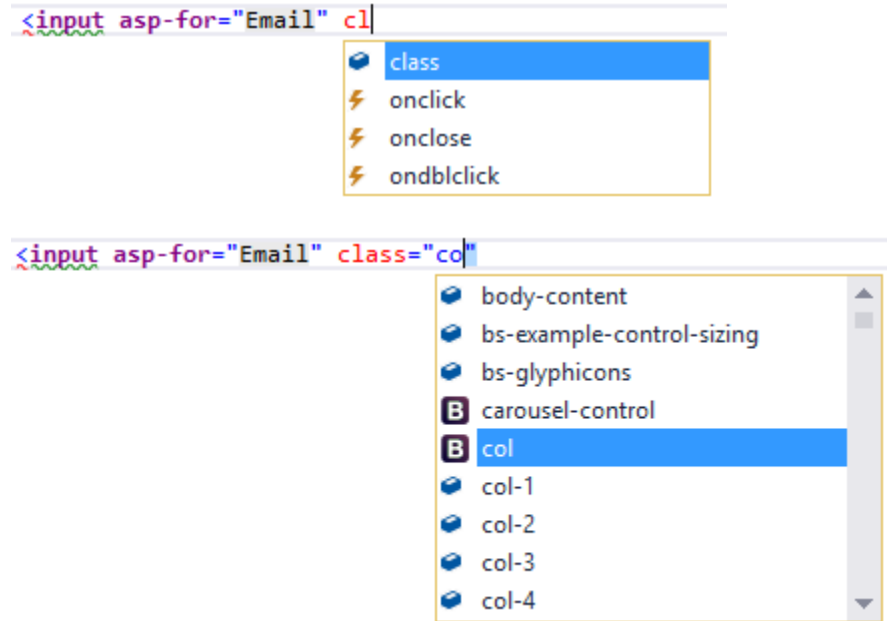
Tag Helpers compared to HTML Helpers   Tag Helpers attach to HTML elements in Razor views, while HTML Helpers are invoked as methods interspersed with HTML in Razor views. Consider the following Razor markup, which creates an HTML label with the CSS class "caption":

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

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

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

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

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

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

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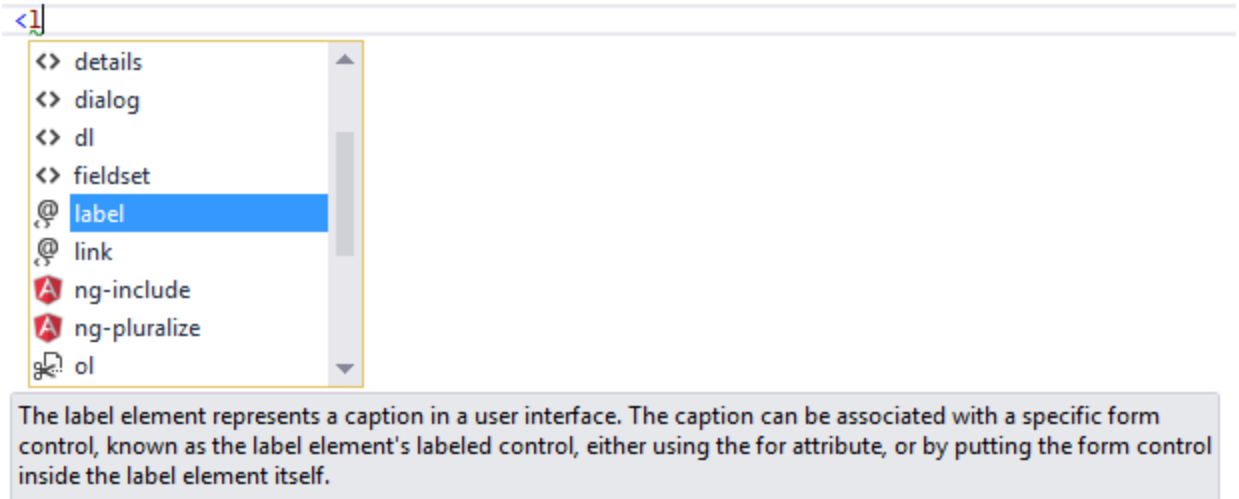
IntelliSense helps you write the entire line. The `LabelTagHelper` also defaults to setting the content of the `asp-for` attribute value ("FirstName") to "First Name"; It converts camel-cased properties to a sentence composed of the property name with a space where each new upper-case letter occurs.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

generates:

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

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

generates:

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

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

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

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

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

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

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

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

Consider the *Email* group:

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

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

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

Tag Helpers compared to Web Server Controls

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

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

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

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

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

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

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

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

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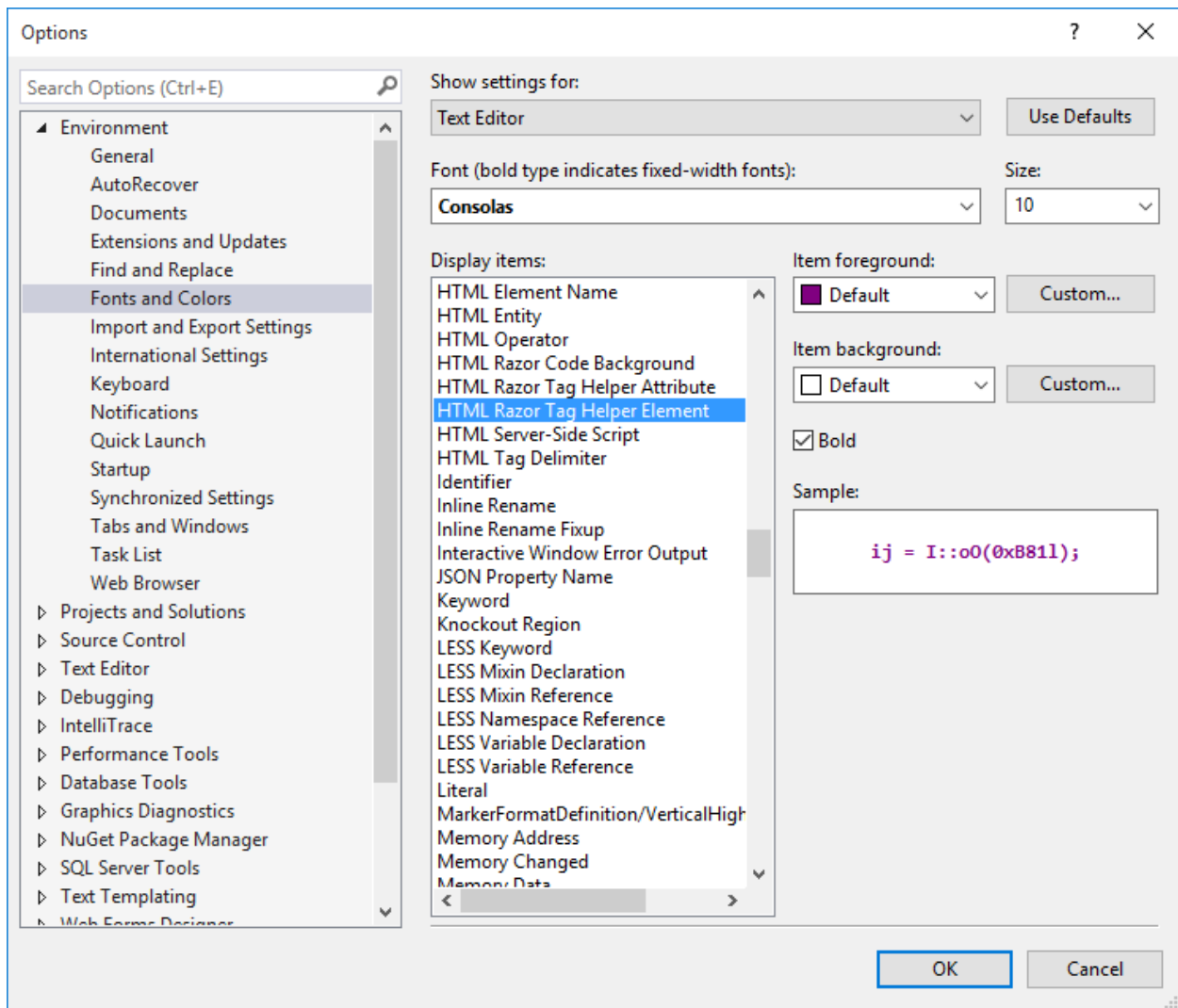
Additional Resources

- *Authoring Tag Helpers*
- *Working with Forms (Tag Helpers)*
- TagHelperSamples on GitHub contains Tag Helper samples for working with Bootstrap.

Authoring Tag Helpers

By Rick Anderson

Sections:

- *Getting started with Tag Helpers*
- *Starting the email Tag Helper*
- *A working email Tag Helper*
- *The bold Tag Helper*
- *Web site information Tag Helper*
- *Condition Tag Helper*
- *Inspecting and retrieving child content*

View or download sample code

Getting started with Tag Helpers   This tutorial provides an introduction to programming Tag Helpers. *Introduction to Tag Helpers* describes the benefits that Tag Helpers provide.

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

1. Create a new ASP.NET Core project called **AuthoringTagHelpers**. You won't need authentication for this project.

2. Create a folder to hold the Tag Helpers called *TagHelpers*. The *TagHelpers* folder is *not* required, but it is a reasonable convention. Now let's get started writing some simple tag helpers.

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

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

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

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

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

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

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

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

Notes:

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

- The `EmailTagHelper` class derives from `TagHelper`. The `TagHelper` class provides methods and properties for writing Tag Helpers.

- The overridden `Process` method controls what the tag helper does when executed. The `TagHelper` class also provides an asynchronous version (`ProcessAsync`) with the same parameters.

- The context parameter to `Process` (and `ProcessAsync`) contains information associated with the execution of the current HTML tag.

- The output parameter to `Process` (and `ProcessAsync`) contains a stateful HTML element representative of the original source used to generate an HTML tag and content.

- Our class name has a suffix of **TagHelper**, which is *not* required, but it's considered a best practice convention. You could declare the class as:

```
public class Email : TagHelper
```

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

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

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

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper "AuthoringTagHelpers.TagHelpers3.EmailTagHelper, AuthoringTagHelpers"
```

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

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

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

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

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

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

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

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

Update the `EmailTagHelper` class with the following:

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

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

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

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

Notes:

- Pascal-cased class and property names for tag helpers are translated into their lower kebab case. Therefore, to use the `MailTo` attribute, you'll use `<email mail-to="value"/>` equivalent.

- The last line sets the completed content for our minimally functional tag helper.

- The highlighted line shows the syntax for adding attributes:

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

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

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

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

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

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

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

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

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

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

```csharp
[HtmlTargetElement("email", TagStructure = TagStructure.WithoutEndTag)]
public class EmailVoidTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    // Code removed for brevity
```

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

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

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

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

Notes:

- This version uses the asynchronous `ProcessAsync` method. The asynchronous `GetChildContentAsync` returns a `Task` containing the `TagHelperContent`.

- We use the `output` parameter to get contents of the HTML element.

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

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

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

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

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

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

The bold Tag Helper

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

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

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

/*
 * public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";
    return View("AboutBoldOnly");
    // return View();
}
*/
```

Notes:

- The `[HtmlTargetElement]` attribute passes an attribute parameter that specifies that any HTML element that contains an HTML attribute named "bold" will match, and the `Process` override method in the class will run. In our sample, the `Process` method removes the "bold" attribute and surrounds the containing markup with `<strong></strong>`.

- Because we don't want to replace the existing tag content, we must write the opening `<strong>` tag with the `PreContent.SetHtmlContent` method and the closing `</strong>` tag with the `PostContent.SetHtmlContent` method.

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

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

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

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

3. Run the app. You can use your favorite browser to inspect the source and verify the markup.

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

4. Comment out the [HtmlTargetElement] attribute line and it will default to targeting <bold> tags, that is, HTML markup of the form <bold>. Remember, the default naming convention will match the class name **Bold**TagHelper to <bold> tags.

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

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

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

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

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

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

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

Web site information Tag Helper

1. Add a *Models* folder.

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

```
using System;

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

```
        public bool Approved { get; set; }
        public int TagsToShow { get; set; }
    }
}
```

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

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

namespace AuthoringTagHelpers.TagHelpers
{
    public class WebsiteInformationTagHelper : TagHelper
    {
        public WebsiteContext Info { get; set; }

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

Notes:

- As mentioned previously, tag helpers translates Pascal-cased C# class names and properties for tag helpers into lower kebab case. Therefore, to use the `WebsiteInformationTagHelper` in Razor, you'll write `<website-information />`.

- We are not explicitly identifying the target element with the `[HtmlTargetElement]` attribute, so the default of `website-information` will be targeted. If you applied the following attribute (note it's not kebab case but matches the class name):

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

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

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

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

- The `$` (dollar sign) in the following line uses an interpolated string:

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

5. Add the following markup to the *About.cshtml* view. The highlighted markup displays the web site information.

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

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

<bold> Is this bold?</bold>

<h3> web site info </h3>
<website-information info="new WebsiteContext {
                                    Version = new Version(1, 3),
                                    CopyrightYear = 1638,
                                    Approved = true,
                                    TagsToShow = 131 }" />
```

Note: In the Razor markup shown below:

```
<website-information info="new WebsiteContext {
                                    Version = new Version(1, 3),
                                    CopyrightYear = 1638,
                                    Approved = true,
                                    TagsToShow = 131 }" />
```

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

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

Note:

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

```
<website-information info="new WebsiteContext {
                                    Version = new Version(1, 3),
                                    CopyrightYear = 1638,
                                    Approved = true,
                                    TagsToShow = 131 }" >
</website-information>
```

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

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

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

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

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

```
            {
                output.SuppressOutput();
            }
        }
    }
}
```

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

```
@using AuthoringTagHelpers.Models
@model WebsiteContext

@{
    ViewData["Title"] = "Home Page";
}

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

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

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

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

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

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

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

```
        }
}
```

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

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

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

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

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

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

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

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

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

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

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

3. Run the app and verify that the tag helper renders the anchor correctly.

4. Update the `AutoLinker` class to include the `AutoLinkerWwwTagHelper` which will convert www text to an anchor tag that also contains the original www text. The updated code is highlighted below:

```
[HtmlTargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
```

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

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

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

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

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

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

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent,
            @"\b(www\.)(\S+)\b",
```

```
                "<a target=\"_blank\" href=\"http://$0\">$0</a>"));   // www version
        }
    }
}
```

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

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

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

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

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

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

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

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

Inspecting and retrieving child content The tag-helpers provide several properties to retrieve content.

- The result of GetChildContentAsync can be appended to output.Content.

- You can inspect the result of GetChildContentAsync with GetContent.

- If you modify output.Content, the TagHelper body will not be executed or rendered unless you call GetChildContentAsync as in our auto-linker sample:

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

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

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

Partial Views

By Steve Smith

ASP.NET Core MVC supports partial views, which are useful when you have reusable parts of web pages you want to share between different views.

Sections:

- *What are Partial Views?*
- *When Should I Use Partial Views?*
- *Declaring Partial Views*
- *Referencing a Partial View*
- *Accessing Data From Partial Views*

View or download sample code

What are Partial Views?

A partial view is a view that is rendered within another view. The HTML output generated by executing the partial view is rendered into the calling (or parent) view. Like views, partial views use the *.cshtml* file extension.

: If you're coming from an ASP.NET Web Forms background, partial views are similar to user controls.

When Should I Use Partial Views?

Partial views are an effective way of breaking up large views into smaller components. They can reduce duplication of view content and allow view elements to be reused. Common layout elements should be specified in *_Layout.cshtml*. Non-layout reusable content can be encapsulated into partial views.

If you have a complex page made up of several logical pieces, it can be helpful to work with each piece as its own partial view. Each piece of the page can be viewed in isolation from the rest of the page, and the view for the page itself becomes much simpler since it only contains the overall page structure and calls to render the partial views.

: Follow the Don't Repeat Yourself Principle in your views.

Declaring Partial Views

Partial views are created like any other view: you create a *.cshtml* file within the *Views* folder. There is no semantic difference between a partial view and a regular view - they are just rendered differently. You can have a view that is returned directly from a controller's `ViewResult`, and the same view can be used as a partial view. The main difference between how a view and a partial view are rendered is that partial views do not run *_ViewStart.cshtml* (while views do - learn more about *_ViewStart.cshtml* in *Layout*).

Referencing a Partial View

From within a view page, there are several ways in which you can render a partial view. The simplest is to use `Html.Partial`, which returns an `IHtmlString` and can be referenced by prefixing the call with `@`:

```
@Html.Partial("AuthorPartial")
```

The `PartialAsync` method is available for partial views containing asynchronous code (although code in views is generally discouraged):

```
@await Html.PartialAsync("AuthorPartial")
```

You can render a partial view with `RenderPartial`. This method doesn't return a result; it streams the rendered output directly to the response. Because it doesn't return a result, it must be called within a Razor code block (you can also call `RenderPartialAsync` if necessary):

```
@{
    Html.RenderPartial("AuthorPartial");
}
```

Because it streams the result directly, `RenderPartial` and `RenderPartialAsync` may perform better in some scenarios. However, in most cases it's recommended you use `Partial` and `PartialAsync`.

: If your views need to execute code, the recommended pattern is to use a *view component* instead of a partial view.

Partial View Discovery   When referencing a partial view, you can refer to its location in several ways:

```
// Uses a view in current folder with this name
// If none is found, searches the Shared folder
@Html.Partial("ViewName")

// A view with this name must be in the same folder
@Html.Partial("ViewName.cshtml")

// Locate the view based on the application root
// Paths that start with "/" or "~/" refer to the application root
@Html.Partial("~/Views/Folder/ViewName.cshtml")
@Html.Partial("/Views/Folder/ViewName.cshtml")

// Locate the view using relative paths
@Html.Partial("../Account/LoginPartial.cshtml")
```

If desired, you can have different partial views with the same name in different view folders. When referencing the views by name (without file extension), views in each folder will use the partial view in the same folder with them. You can also specify a default partial view to use, placing it in the *Shared* folder. This view will be used by any views that don't have their own copy of the partial view in their folder. In this way, you can have a default partial view (in *Shared*), which can be overridden by a partial view with the same name in the same folder as the parent view.

Partial views can be *chained*. That is, a partial view can call another partial view (as long as you don't create a loop). Within each view or partial view, relative paths are always relative to that view, not the root or parent view.

: If you declare a *Razor* `section` in a partial view, it will not be visible to its parent(s); it will be limited to the partial view.

Accessing Data From Partial Views

When a partial view is instantiated, it gets a copy of the parent view's `ViewData` dictionary. Updates made to the data within the partial view are not persisted to the parent view. `ViewData` changed in a partial view is lost when the partial view returns.

You can pass an instance of `ViewDataDictionary` to the partial view:

```
@Html.Partial("PartialName", customViewData)
```

You can also pass a model into a partial view. This can be the page's view model, or some portion of it, or a custom object. Simply pass in the model as the second parameter when calling `Partial`/`PartialAsync` or `RenderPartial`/`RenderPartialAsync`:

```
@Html.Partial("PartialName", viewModel)
```

You can pass an instance of `ViewDataDictionary` and a view model to a partial view:

```
@Html.Partial("PartialName", viewModel, customViewData)
```

An Example   The following view specifies a view model of type `Article`. `Article` has an `AuthorName` property that is passed to a partial view named *AuthorPartial*, and a property of type `List<ArticleSection>`, which is passed (in a loop) to a partial devoted to rendering that type:

```
@using PartialViewsSample.ViewModels
@model Article

<h2>@Model.Title</h2>
@Html.Partial("AuthorPartial", Model.AuthorName)
@Model.PublicationDate

@foreach (var section in @Model.Sections)
{
    @Html.Partial("ArticleSection", section)
}
```

The *AuthorPartial* (which in this case is in the */Views/Shared* folder):

```
@model string
<div>
    <h3>@Model</h3>
    This partial view came from /Views/Shared/AuthorPartial.cshtml.<br/>
</div>
```

The *ArticleSection* partial:

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title</h3>
<div>
    @Model.Content
</div>
```

At runtime, the partials are rendered into the parent view, which itself is rendered within the shared *_Layout.cshtml*, resulting in output like this:
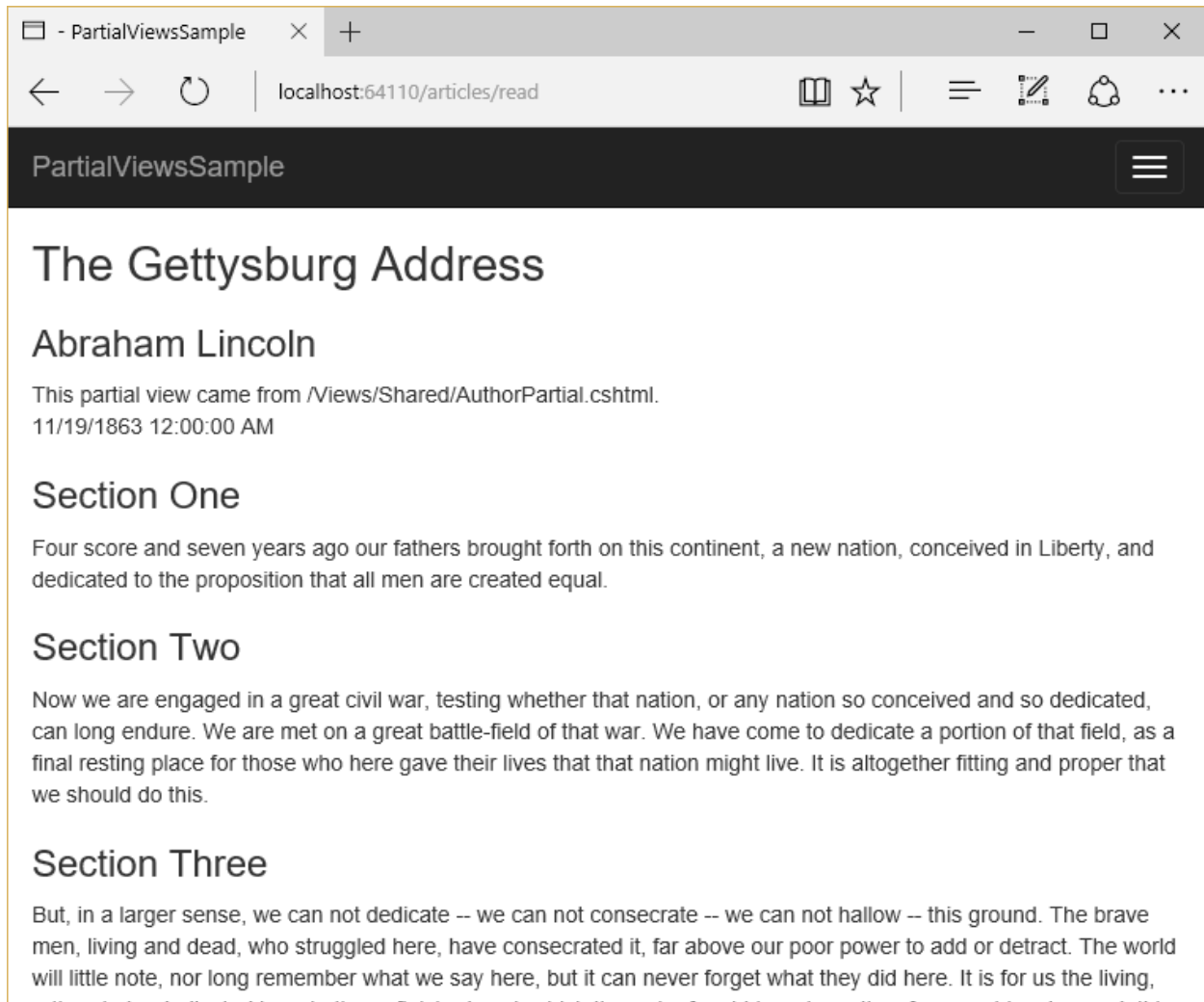
Injecting Services Into Views

By Steve Smith

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

Sections:

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

View or download sample code

A Simple Example

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

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

An example of `@inject` in action:

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

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

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

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

```
using System.Linq;
using ViewInjectSample.Interfaces;

namespace ViewInjectSample.Model.Services
{
    public class StatisticsService
    {
        private readonly IToDoItemRepository _toDoItemRepository;

        public StatisticsService(IToDoItemRepository toDoItemRepository)
        {
            _toDoItemRepository = toDoItemRepository;
        }

        public int GetCount()
        {
            return _toDoItemRepository.List().Count();
        }

        public int GetCompletedCount()
        {
            return _toDoItemRepository.List().Count(x => x.IsDone);
        }

        public double GetAveragePriority()
        {
            if (_toDoItemRepository.List().Count() == 0)
            {
                return 0.0;
            }

            return _toDoItemRepository.List().Average(x => x.Priority);
        }
    }
}
```

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

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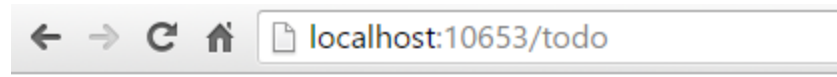
Populating Lookup Data

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

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

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

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

Update Profile

Name: Steve
Gender: Male
State: Ohio
Fav. Color: Blue
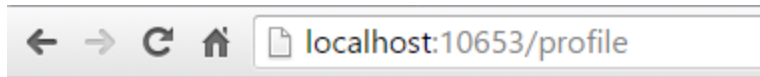
These lists are populated by a service that has been injected into the view:

```
@using System.Threading.Tasks
@using ViewInjectSample.Model.Services
@model ViewInjectSample.Model.Profile
@inject ProfileOptionsService Options
<!DOCTYPE html>
<html>
<head>
    <title>Update Profile</title>
</head>
<body>
<div>
    <h1>Update Profile</h1>
    Name: @Html.TextBoxFor(m => m.Name)
    <br/>
    Gender: @Html.DropDownList("Gender",
            Options.ListGenders().Select(g =>
                new SelectListItem() { Text = g, Value = g }))
    <br/>

    State: @Html.DropDownListFor(m => m.State.Code,
            Options.ListStates().Select(s =>
                new SelectListItem() { Text = s.Name, Value = s.Code}))
    <br />

    Fav. Color: @Html.DropDownList("FavColor",
            Options.ListColors().Select(c =>
                new SelectListItem() { Text = c, Value = c }))
    </div>
</body>
</html>
```

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

```
using System.Collections.Generic;

namespace ViewInjectSample.Model.Services
{
    public class ProfileOptionsService
    {
        public List<string> ListGenders()
```

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

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

Overriding Services

In addition to injecting new services, this technique can also be used to override previously injected services on a page. The figure below shows all of the fields available on the page used in the first example:
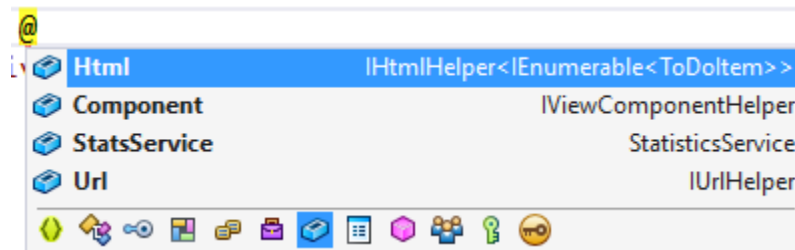
As you can see, the default fields include `Html`, `Component`, and `Url` (as well as the `StatsService` that we injected). If for instance you wanted to replace the default HTML Helpers with your own, you could easily do so using `@inject`:

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

```
11          Test: @Html.Value
12      </div>
13  </body>
14  </html>
```

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

See Also

- Simon Timms Blog: Getting Lookup Data Into Your View

View Components

By Rick Anderson

> **Sections:**
>
> - *Introducing view components*
> - *Creating a view component*
> - *Invoking a view component*
> - *Walkthrough: Creating a simple view component*
> - *Additional Resources*

View or download sample code

Introducing view components

New to ASP.NET Core MVC, view components are similar to partial views, but they are much more powerful. View components don't use model binding, and only depend on the data you provide when calling into it. A view component:

- Renders a chunk rather than a whole response

- Includes the same separation-of-concerns and testability benefits found between a controller and view

- Can have parameters and business logic

- Is typically invoked from a layout page

View Components are intended anywhere you have reusable rendering logic that is too complex for a partial view, 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

- A login panel that would be rendered on every page and show either the links to log out or log in, depending on the log in state of the user

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

Creating a view component

This section contains the high level requirements to create a view component. Later in the article we'll examine each step in detail and create a view component.

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

- Deriving from *ViewComponent*

- Decorating a class with the `[ViewComponent]` attribute, or deriving from a class with the `[ViewComponent]` attribute

- Creating a class where the name ends with the suffix *ViewComponent*

Like controllers, view components must be public, non-nested, and non-abstract classes. The view component name is the class name with the "ViewComponent" suffix removed. It can also be explicitly specified using the ViewComponentAttribute.Name property.

A view component class:

- Fully supports constructor *dependency injection*

- Does not take part in the controller lifecycle, which means you can't use *filters* in a view component

View component methods   A view component defines its logic in an `InvokeAsync` method that returns an IViewComponentResult. Parameters come directly from invocation of the view component, not from model binding. A view component never directly handles a request. Typically a view component initializes a model and passes it to a view by calling the View method. In summary, view component methods:

- Define an *InvokeAsync'* method that returns an `IViewComponentResult`

- Typically initializes a model and passes it to a view by calling the ViewComponent View method

- Parameters come from the calling method, not HTTP, there is no model binding

- Are not reachable directly as an HTTP endpoint, they are invoked from your code (usually in a view). A view component never handles a request

- Are overloaded on the signature rather than any details from the current HTTP request

View search path   The runtime searches for the view in the following paths:

- Views/<controller_name>/Components/<view_component_name>/<view_name>

- Views/Shared/Components/<view_component_name>/<view_name>

The default view name for a view component is *Default*, which means your view file will typically be named *Default.cshtml*. You can specify a different view name when creating the view component result or when calling the `View` method.

We recommend you name the view file *Default.cshtml* and use the *Views/Shared/Components/<view_component_name>/<view_name>* path.   The   `PriorityList`   view   component   used   in   this   sample   uses *Views/Shared/Components/PriorityList/Default.cshtml* for the view component view.

Invoking a view component

To use the view component, call `@Component.InvokeAsync("Name of view component", <anonymous type containing parameters>)` from a view. The parameters will be passed to the `InvokeAsync` method. The `PriorityList` view component developed in the article is invoked from the *Views/Todo/Index.cshtml* view file. In the following, the `InvokeAsync` method is called with two parameters:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 2, isDone = false })
```

Invoking a view component directly from a controller   View components are typically invoked from a view, but you can invoke them directly from a controller method. While view components do not define endpoints like controllers, you can easily implement a controller action that returns the content of a ViewComponentResult.

In this example, the view component is called directly from the controller:

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```

Walkthrough: Creating a simple view component

Download, build and test the starter code. It's a simple project with a `Todo` controller that displays a list of *Todo* items.
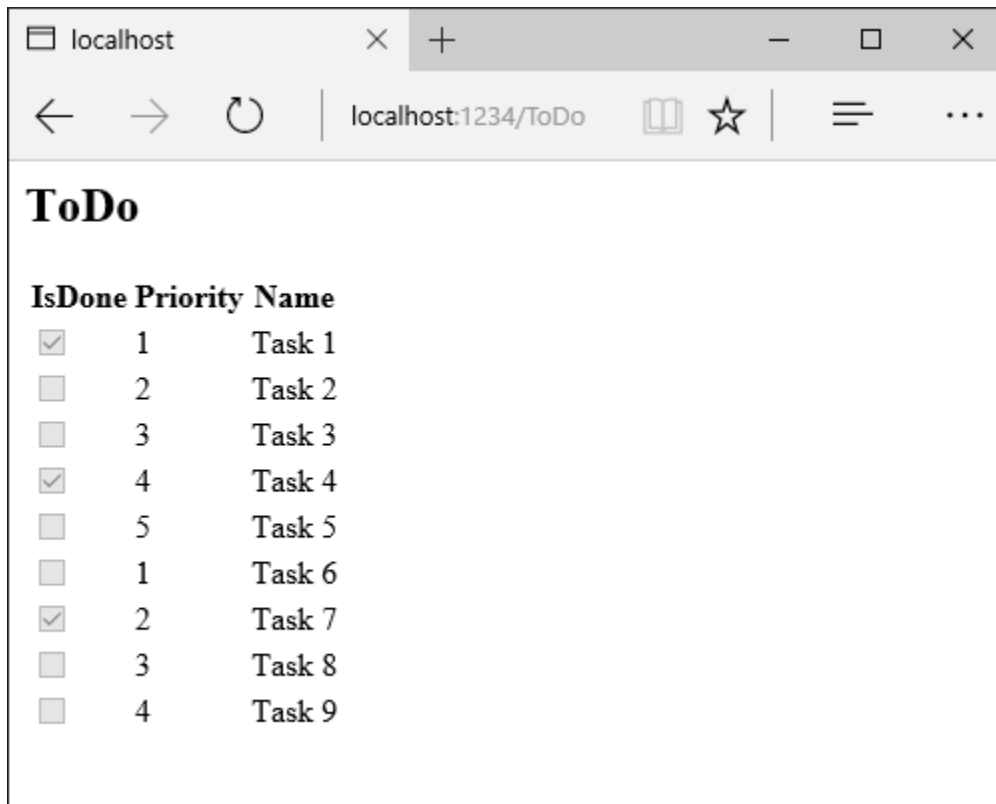
Add a ViewComponent class Create a *ViewComponents* folder and add the following `PriorityListViewComponent` class.

```
using Microsoft.AspNet.Mvc;
using Microsoft.Data.Entity;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityListViewComponent : ViewComponent
    {
        private readonly ToDoContext db;

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

        public async Task<IViewComponentResult> InvokeAsync(
        int maxPriority, bool isDone)
        {
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(items);
        }
        private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
            return db.ToDo.Where(x => x.IsDone == isDone &&
                                 x.Priority <= maxPriority).ToListAsync();
        }
    }
}
```

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 run-time will use the string "PriorityList" when referencing the class component from a view. I'll explain that in more detail later.

- The `[ViewComponent]` attribute can change the name used to reference a view component. For example, we could have named the class `XYZ`, and applied the `ViewComponent` attribute:

```
[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 *dependency injection* to make the data context available.

- `InvokeAsync` exposes a method which can be called from a view, and it can take an arbitrary number of arguments.

- The `InvokeAsync` method returns the set of `ToDo` items that are not completed and have priority lower than or equal to `maxPriority`.

Create the view component Razor view

1. Create the *Views/Shared/Components* folder. This folder **must** be named *Components*.

2. Create the *Views/Shared/Components/PriorityList* folder. This folder name must match the name of the view component class, or the name of the class minus the suffix (if we followed convention and used the *ViewComponent* suffix in the class name). If you used the `ViewComponent` attribute, the class name would need to match the attribute designation.

3. Create a *Views/Shared/Components/PriorityList/Default.cshtml* Razor view.

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

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

The Razor view takes a list of `TodoItem` and displays them. If the view component `InvokeAsync` 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. To override the default styling for a specific controller, add a view to the controller specific view folder (for example *Views/Todo/Components/PriorityList/Default.cshtml)*.

If the view component was controller specific, you could add it to the controller specific folder (*Views/Todo/Components/PriorityList/Default.cshtml*)

4. Add a `div` containing a call to the priority list component to the bottom of the *Views/Todo/index.cshtml* file:

```
    }
</table>
<div >
    @await Component.InvokeAsync("PriorityList", new { maxPriority = 2, isDone = false })
</div>
```

The markup `@Component.InvokeAsync` 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. `InvokeAsync` can take an arbitrary number of arguments.

The following image shows the priority items:
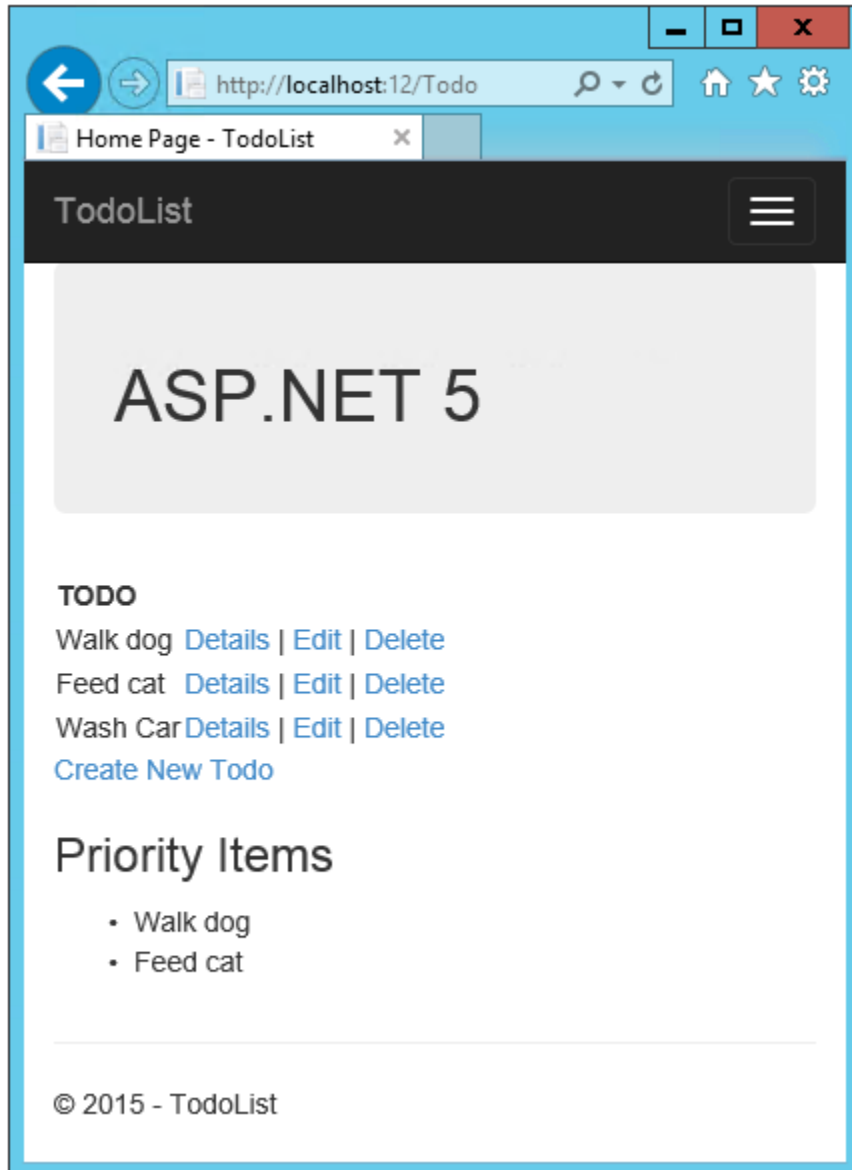
You can also call the view component directly from the controller:

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```

Specifying a view name   A complex view component might need to specify a non-default view under some conditions. The following code 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);
}
```

Copy the *Views/Shared/Components/PriorityList/Default.cshtml* file to a view named *Views/Shared/Components/PriorityList/PVC.cshtml*. Add a heading to indicate the PVC view is being used.

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

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

Update *Views/TodoList/Index.cshtml*

```
</table>

<div>
    @await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
</div>
```

Run the app and verify PVC view.
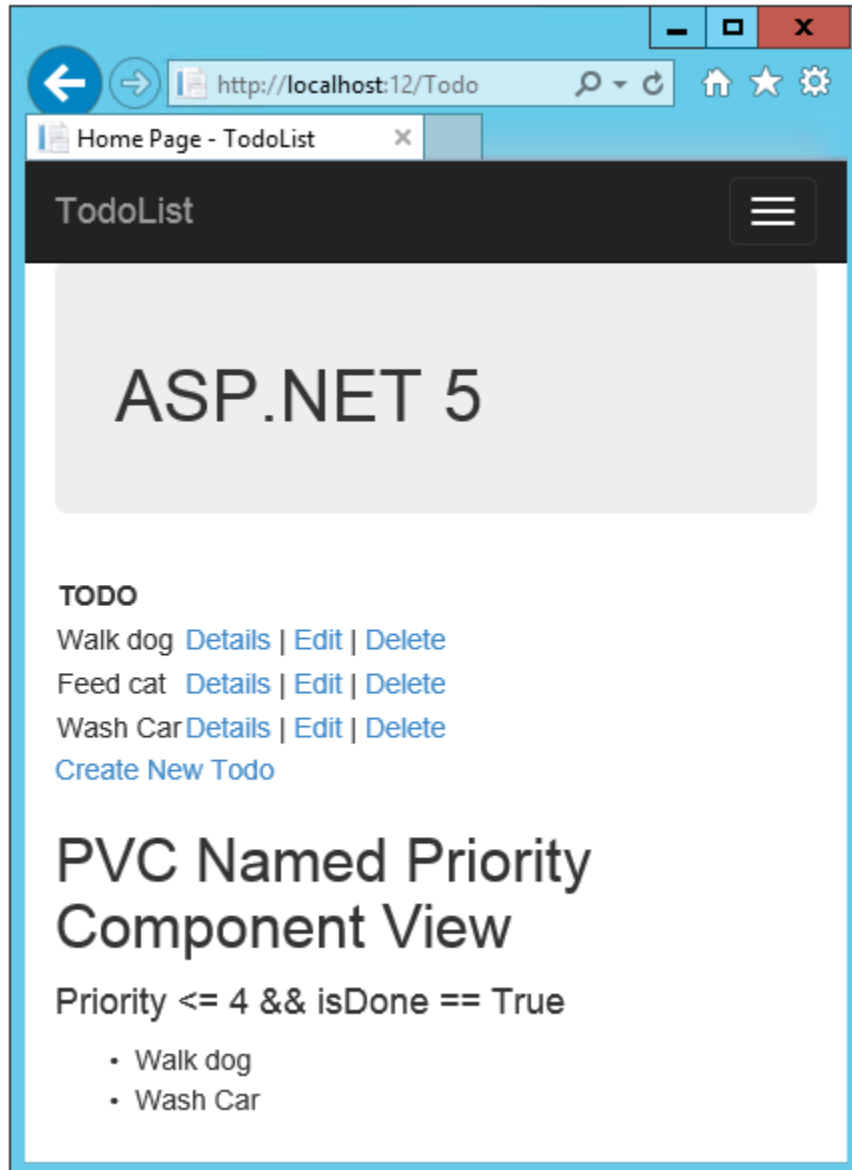
If the PVC view is not rendered, verify you are calling the view component with a priority of 4 or higher.

Examine the view path

1. Change the priority parameter to three or less so the priority view is not returned.

2. Temporarily rename the *Views/Todo/Components/PriorityList/Default.cshtml* to *Temp.cshtml*.

3. Test the app, you'll get the following error:

```
An unhandled exception occurred while processing the request.

InvalidOperationException: The view 'Components/PriorityList/Default'
    was not found. The following locations were searched:
    /Views/ToDo/Components/PriorityList/Default.cshtml
    /Views/Shared/Components/PriorityList/Default.cshtml.
Microsoft.AspNetCore.Mvc.ViewEngines.ViewEngineResult.EnsureSuccessful()
```

4. Copy *Views/Shared/Components/PriorityList/Default.cshtml to *Views/Todo/Components/PriorityList/Default.cshtml*.

5. Add some markup to the *Todo* view component view to indicate the view is from the *Todo* folder.

6. Test the **non-shared** component view.
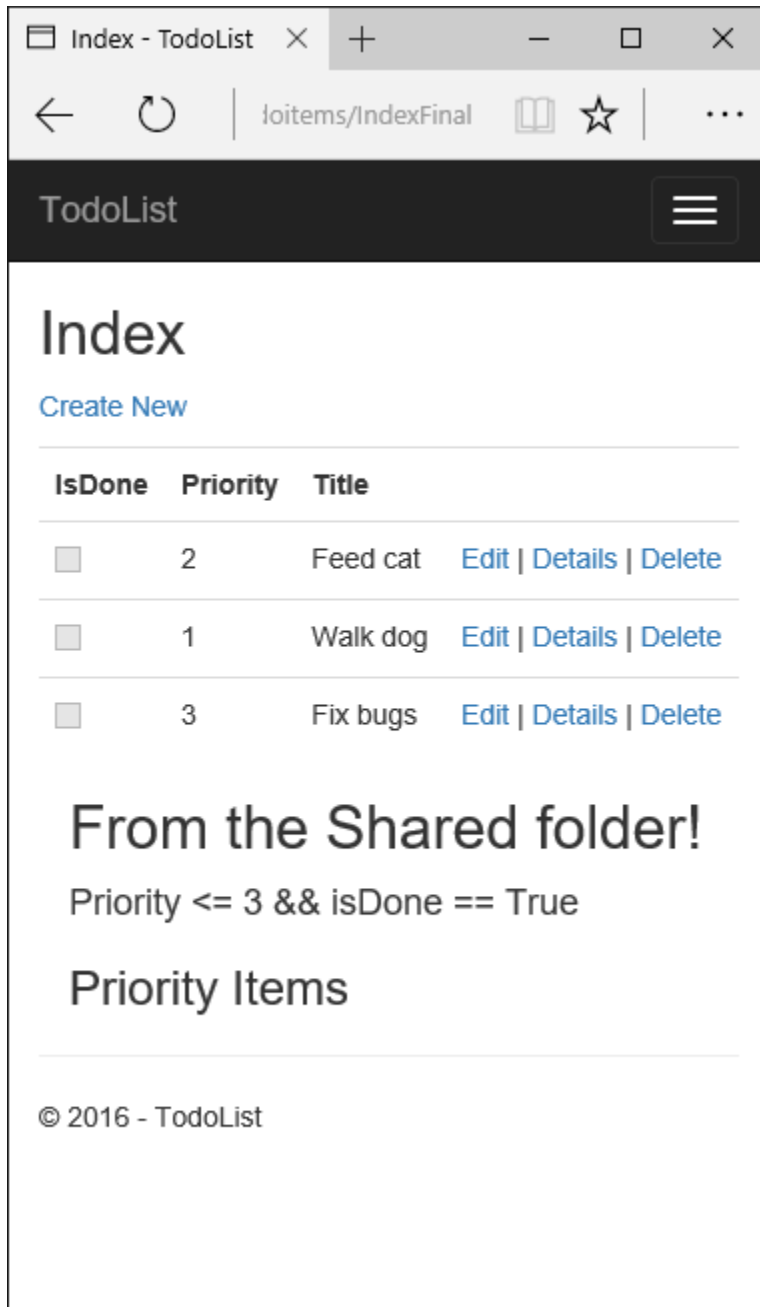
Avoiding magic strings If you want compile time safety you can replace the hard coded view component name with the class name. Create the view component without the "ViewComponent" suffix:

```
using Microsoft.AspNet.Mvc;
using Microsoft.Data.Entity;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityList : ViewComponent
    {
        private readonly ToDoContext db;

        public PriorityList(ToDoContext context)
        {
            db = context;
        }

        public async Task<IViewComponentResult> InvokeAsync(
        int maxPriority, bool isDone)
        {
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(items);
        }
        private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
            return db.ToDo.Where(x => x.IsDone == isDone &&
                                 x.Priority <= maxPriority).ToListAsync();
        }
    }
}
```

Add a `using` statement to your Razor view file and use the `nameof` operator:

```
@using ViewComponentSample.Models
@using ViewComponentSample.ViewComponents
@model IEnumerable<TodoItem>

<h2>ToDo nameof</h2>
<!-- Markup removed for brevity.  -->
    }
</table>

<div>

    @await Component.InvokeAsync(nameof(PriorityList), new { maxPriority = 4, isDone = true })
</div>
```

Additional Resources

- *Injecting Services Into Views*
- ViewComponent

Creating a Custom View Engine

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

Building Mobile Specific Views

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.5.4 Controllers

Controllers, Actions, and Action Results

By Steve Smith

Actions and action results are a fundamental part of how developers build apps using ASP.NET MVC.

> **Sections:**
>
> - *What is a Controller*
> - *Defining Actions*

What is a Controller

In ASP.NET MVC, a *Controller* is used to define and group a set of actions. An *action* (or *action method*) is a method on a controller that handles incoming requests. Controllers provide a logical means of grouping similar actions together, allowing common sets of rules (e.g. routing, caching, authorization) to be applied collectively. Incoming requests are mapped to actions through *routing*.

In ASP.NET Core MVC, a controller can be any instantiable class that ends in "Controller" or inherits from a class that ends with "Controller". Controllers should follow the Explicit Dependencies Principle and request any dependencies their actions require through their constructor using *dependency injection*.

By convention, controller classes:

- Are located in the root-level "Controllers" folder

- Inherit from Microsoft.AspNetCore.Mvc.Controller

These two conventions are not required.

Within the Model-View-Controller pattern, a Controller is responsible for the initial processing of the request and instantiation of the Model. Generally, business decisions should be performed within the Model.

: The Model should be a *Plain Old CLR Object (POCO)*, not a `DbContext` or database-related type.

The controller takes the result of the model's processing (if any), returns the proper view along with the associated view data. Learn more: *Overview of ASP.NET MVC* and *Getting started with ASP.NET Core MVC and Visual Studio*.

: The Controller is a *UI level* abstraction. Its responsibility is to ensure incoming request data is valid and to choose which view (or result for an API) should be returned. In well-factored apps it will not directly include data access or business logic, but instead will delegate to services handling these responsibilities.

Defining Actions

Any public method on a controller type is an action. Parameters on actions are bound to request data and validated using *model binding*.

> **:** Action methods that accept parameters should verify the `ModelState.IsValid` property is true.

Action methods should contain logic for mapping an incoming request to a business concern. Business concerns should typically be represented as services that your controller accesses through *dependency injection*. Actions then map the result of the business action to an application state.

Actions can return anything, but frequently will return an instance of `IActionResult` (or `Task<IActionResult>` for async methods) that produces a response. The action method is responsible for choosing *what kind of response*; the action result *does the responding*.

Controller Helper Methods   Although not required, most developers will want to have their controllers inherit from the base `Controller` class. Doing so provides controllers with access to many properties and helpful methods, including the following helper methods designed to assist in returning various responses:

View Returns a view that uses a model to render HTML. Example: `return View(customer);`

HTTP Status Code Return an HTTP status code. Example: `return BadRequest();`

Formatted Response Return `Json` or similar to format an object in a specific manner. Example: `return Json(customer);`

Content negotiated response Instead of returning an object directly, an action can return a content negotiated response (using `Ok`, `Created`, `CreatedAtRoute` or `CreatedAtAction`). Examples: `return Ok();` or `return CreatedAtRoute("routename",values,newobject");`

Redirect Returns a redirect to another action or destination (using `Redirect`,``LocalRedirect``,``RedirectToAction`` or `RedirectToRoute`). Example: `return RedirectToAction("Complete", new {id = 123});`

In addition to the methods above, an action can also simply return an object. In this case, the object will be formatted based on the client's request. Learn more about *Formatting Response Data*

Cross-Cutting Concerns   In most apps, many actions will share parts of their workflow. For instance, most of an app might be available only to authenticated users, or might benefit from caching. When you want to perform some logic before or after an action method runs, you can use a *filter*. You can help keep your actions from growing too large by using *Filters* to handle these cross-cutting concerns. This can help eliminate duplication within your actions, allowing them to follow the Don't Repeat Yourself (DRY) principle.

In the case of authorization and authentication, you can apply the `Authorize` attribute to any actions that require it. Adding it to a controller will apply it to all actions within that controller. Adding this attribute will ensure the appropriate filter is applied to any request for this action. Some attributes can be applied at both controller and action levels to provide granular control over filter behavior. Learn more: *Filters* and *Authorization Filters*.

Other examples of cross-cutting concerns in MVC apps may include:

- *Error handling*

- *Response Caching*

: Many cross-cutting concerns can be handled using filters in MVC apps. Another option to keep in mind that is available to any ASP.NET Core app is custom *middleware*.

Routing to Controller Actions

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

Filters

By Steve Smith

Filters in ASP.NET MVC allow you to run code before or after a particular stage in the execution pipeline. Filters can be configured globally, per-controller, or per-action.

Sections

- *How do filters work?*
- *Configuring Filters*
- *Authorization Filters*
- *Resource Filters*
- *Action Filters*
- *Exception Filters*
- *Result Filters*
- *Filters vs. Middleware*

View or download sample from GitHub.

How do filters work?

Each filter type is executed at a different stage in the pipeline, and thus has its own set of intended scenarios. Choose what type of filter to create based on the task you need it to perform, and where in the request pipeline it executes.

Filters run within the MVC Action Invocation Pipeline, sometimes referred to as the *Filter Pipeline*, which runs after MVC selects the action to execute.
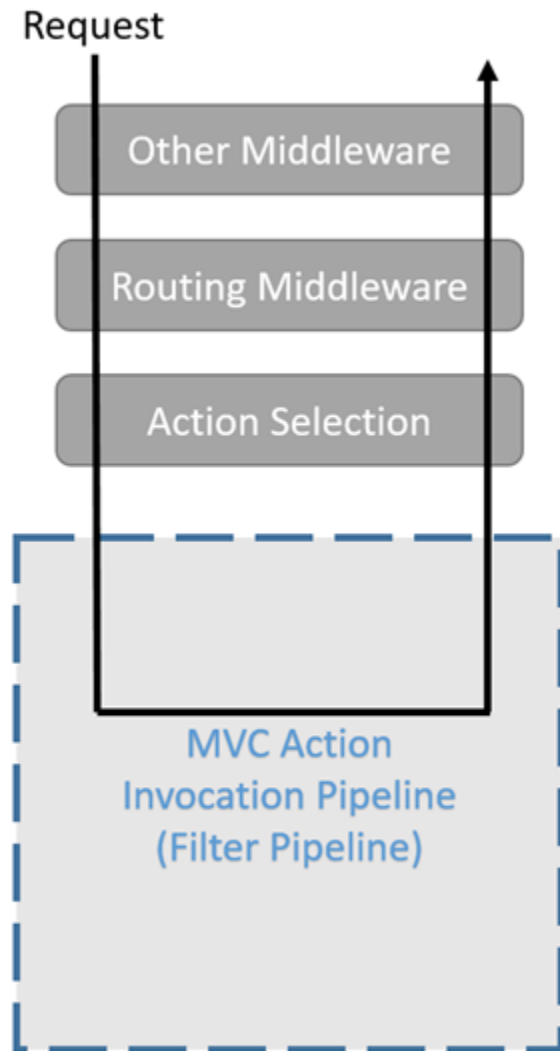
Different filter types run at different points within the pipeline. Some filters, like authorization filters, only run before the next stage in the pipeline, and take no action afterward. Other filters, like action filters, can execute both before and after other parts of the pipeline execute, as shown below.
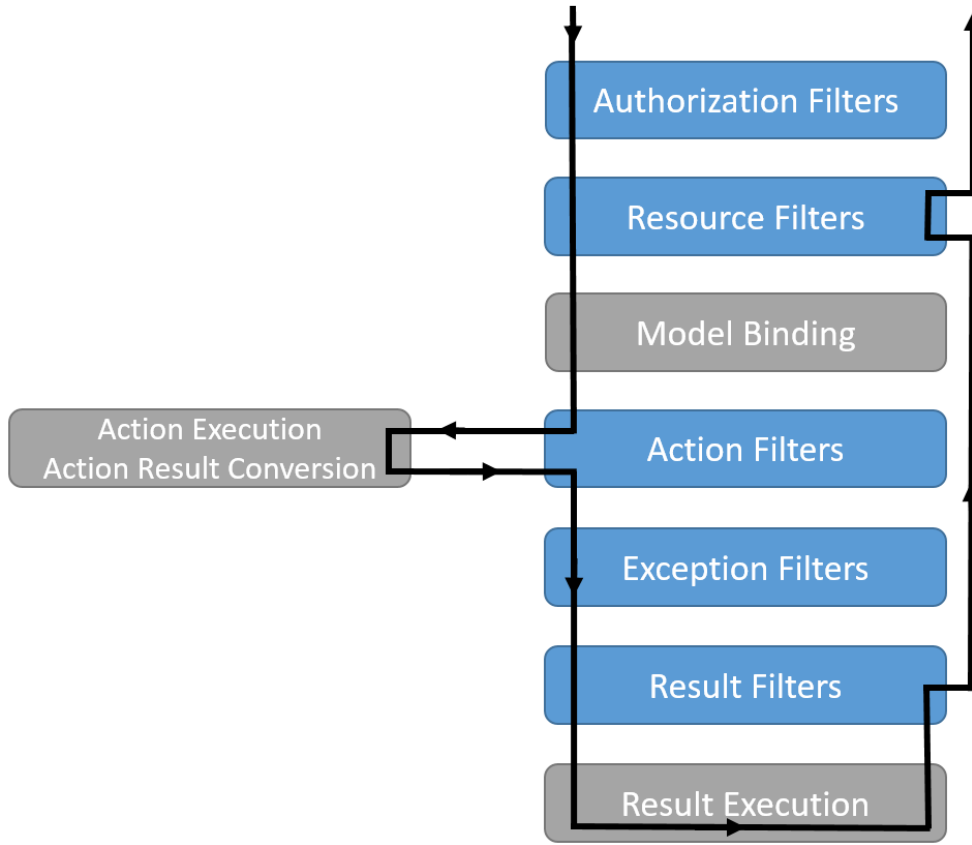
Selecting a Filter   *Authorization filters* are used to determine whether the current user is authorized for the request being made.

Resource filters are the first filter to handle a request after authorization, and the last one to touch the request as it is leaving the filter pipeline. They're especially useful to implement caching or otherwise short-circuit the filter pipeline for performance reasons.

Action filters wrap calls to individual action method calls, and can manipulate the arguments passed into an action as well as the action result returned from it.

Exception filters are used to apply global policies to unhandled exceptions in the MVC app.

Result filters wrap the execution of individual action results, and only run when the action method has executed successfully. They are ideal for logic that must surround view execution or formatter execution.

Implementation   All filters support both synchronous and asynchronous implementations through different interface definitions. Choose the sync or async variant depending on the kind of task you need to perform. They are interchangeable from the framework's perspective.

Synchronous filters define both an On*Stage*Executing and On*Stage*Executed method (with noted exceptions). The On*Stage*Executing method will be called before the event pipeline stage by the Stage name, and the On*Stage*Executed method will be called after the pipeline stage named by the Stage name.

```
using FiltersSample.Helper;
using Microsoft.AspNetCore.Mvc.Filters;
```

```
namespace FiltersSample.Filters
{
    public class SampleActionFilter : IActionFilter
    {
        public void OnActionExecuting(ActionExecutingContext context)
        {
            // do something before the action executes
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            // do something after the action executes
        }
    }
}
```

Asynchronous filters define a single On*Stage*ExecutionAsync method that will surround execution of the pipeline stage named by Stage. The On*Stage*ExecutionAsync method is provided a *Stage*ExecutionDelegate delegate which will execute the pipeline stage named by Stage when invoked and awaited.

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class SampleAsyncActionFilter : IAsyncActionFilter
    {
        public async Task OnActionExecutionAsync(
            ActionExecutingContext context,
            ActionExecutionDelegate next)
        {
            // do something before the action executes
            await next();
            // do something after the action executes
        }
    }
}
```

: You should only implement *either* the synchronous or the async version of a filter interface, not both. If you need to perform async work in the filter, implement the async interface. Otherwise, implement the synchronous interface. The framework will check to see if the filter implements the async interface first, and if so, it will call it. If not, it will call the synchronous interface's method(s). If you were to implement both interfaces on one class, only the async method would be called by the framework. Also, it doesn't matter whether your action is async or not, your filters can be synchronous or async independent of the action.

Filter Scopes Filters can be *scoped* at three different levels. You can add a particular filter to a particular action as an attribute. You can add a filter to all actions within a controller by applying an attribute at the controller level. Or you can register a filter globally, to be run with every MVC action.

Global filters are added in the ConfigureServices method in Startup, when configuring MVC:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
```

```
        options.Filters.Add(typeof(SampleActionFilter)); // by type
        options.Filters.Add(new SampleGlobalActionFilter()); // an instance
    });

    services.AddScoped<AddHeaderFilterWithDi>();
}
```

Filters can be added by type, or an instance can be added. If you add an instance, that instance will be used for every request. If you add a type, it will be type-activated, meaning an instance will be created for each request and any constructor dependencies will be populated by DI. Adding a filter by type is equivalent to `filters.Add(new TypeFilterAttribute(typeof(MyFilter)))`.

It's often convenient to implement filter interfaces as *Attributes*. Filter attributes are applied to controllers and action methods. The framework includes built-in attribute-based filters that you can subclass and customize. For example, the following filter inherits from ResultFilterAttribute, and overrides its `OnResultExecuting` method to add a header to the response.

```
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class AddHeaderAttribute : ResultFilterAttribute
    {
        private readonly string _name;
        private readonly string _value;

        public AddHeaderAttribute(string name, string value)
        {
            _name = name;
            _value = value;
        }

        public override void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                _name, new string[] { _value });
            base.OnResultExecuting(context);
        }
    }
}
```

Attributes allow filters to accept arguments, as shown in the example above. You would add this attribute to a controller or action method and specify the name and value of the HTTP header you wished to add to the response:

```
[AddHeader("Author", "Steve Smith @ardalis")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using developer tools.");
    }
}
```

The result of the `Index` action is shown below - the response headers are displayed on the bottom right.

Examine the headers using developer tools.
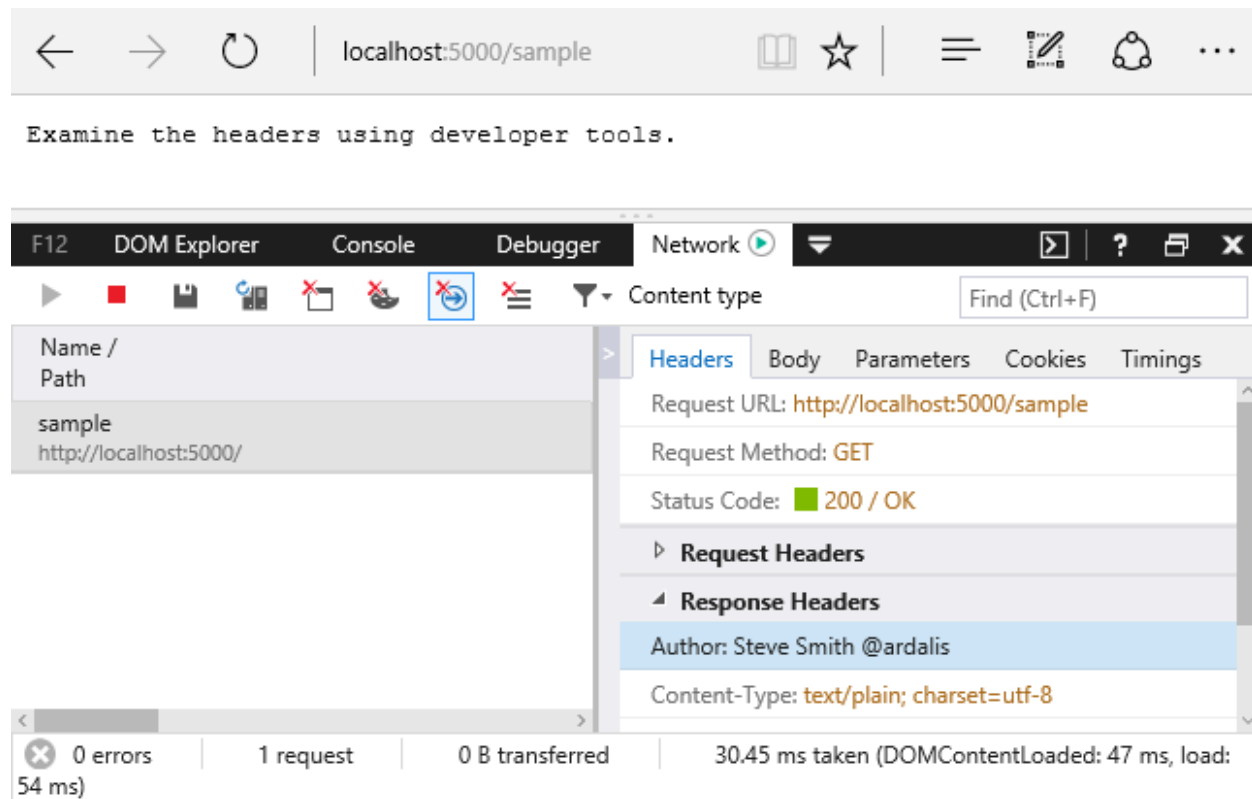
Several of the filter interfaces have corresponding attributes that can be used as base classes for custom implementations.

Filter attributes:

- ActionFilterAttribute
- AuthorizationFilterAttribute
- ExceptionFilterAttribute
- ResultFilterAttribute

Cancellation and Short Circuiting You can short-circuit the filter pipeline at any point by setting the Result property on the context parameter provided to the filter method. For instance, the following ShortCircuitingResourceFilter will prevent any other filters from running later in the pipeline, including any action filters.

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class ShortCircuitingResourceFilterAttribute : Attribute,
            IResourceFilter
    {
        public void OnResourceExecuting(ResourceExecutingContext context)
        {
            context.Result = new ContentResult()
            {
```

```
            Content = "Resource unavailable – header should not be set"
        };
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
}
```

In the following code, both the `ShortCircuitingResourceFilter` and the `AddHeader` filter target the `SomeResource` action method. However, because the `ShortCircuitingResourceFilter` runs first and short-circuits the rest of the pipeline, the `AddHeader` filter never runs for the `SomeResource` action. This behavior would be the same if both filters were applied at the action method level, provided the `ShortCircuitingResourceFilter` ran first (see *Ordering*).

```
[AddHeader("Author", "Steve Smith @ardalis")]
public class SampleController : Controller
{
    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource – header should be set.");
    }
```

Configuring Filters

Global filters are configured within `Startup.cs`. Attribute-based filters that do not require any dependencies can simply inherit from an existing attribute of the appropriate type for the filter in question. To create a filter *without* global scope that requires dependencies from DI, apply the `ServiceFilterAttribute` or `TypeFilterAttribute` attribute to the controller or action.

Dependency Injection Filters that are implemented as attributes and added directly to controller classes or action methods cannot have constructor dependencies provided by *dependency injection* (DI). This is because attributes must have their constructor parameters supplied where they are applied. This is a limitation of how attributes work.

However, if your filters have dependencies you need to access from DI, there are several supported approaches. You can apply your filter to a class or action method using

- `ServiceFilterAttribute`
- `TypeFilterAttribute`
- `IFilterFactory` implemented on your attribute

A `TypeFilter` will instantiate an instance, using services from DI for its dependencies. A `ServiceFilter` retrieves an instance of the filter from DI. The following example demonstrates using a `ServiceFilter`:

```
[ServiceFilter(typeof(AddHeaderFilterWithDi))]
public IActionResult Index()
{
    return View();
}
```

Using `ServiceFilter` without registering the filter type in `ConfigureServices`, throws the following exception:

```
System.InvalidOperationException: No service for type
'FiltersSample.Filters.AddHeaderFilterWithDI' has been registered.
```

To avoid this exception, you must register the `AddHeaderFilterWithDI` type in `ConfigureServices`:

```
services.AddScoped<AddHeaderFilterWithDi>();
```

`ServiceFilterAttribute` implements `IFilterFactory`, which exposes a single method for creating an `IFilter` instance. In the case of `ServiceFilterAttribute`, the `IFilterFactory` interface's `CreateInstance` method is implemented to load the specified type from the services container (DI).

`TypeFilterAttribute` is very similar to `ServiceFilterAttribute` (and also implements `IFilterFactory`), but its type is not resolved directly from the DI container. Instead, it instantiates the type using a `Microsoft.Extensions.DependencyInjection.ObjectFactory`.

> Because of this difference, types that are referenced using the `TypeFilterAttribute` do not need to be registered with the container first (but they will still have their dependencies fulfilled by the container). Also, `TypeFilterAttribute` can optionally accept constructor arguments for the type in question. The following example demonstrates how to pass arguments to a type using `TypeFilterAttribute`:

```
[TypeFilter(typeof(AddHeaderAttribute),
    Arguments = new object[] { "Author", "Steve Smith (@ardalis)" })]
public IActionResult Hi(string name)
{
    return Content($"Hi {name}");
}
```

If you have a simple filter that doesn't require any arguments, but which has constructor dependencies that need to be filled by DI, you can inherit from `TypeFilterAttribute`, allowing you to use your own named attribute on classes and methods (instead of `[TypeFilter(typeof(FilterType))]`). The following filter shows how this can be implemented:

```
public class SampleActionFilterAttribute : TypeFilterAttribute
{
    public SampleActionFilterAttribute():base(typeof(SampleActionFilterImpl))
    {
    }

    private class SampleActionFilterImpl : IActionFilter
    {
        private readonly ILogger _logger;
        public SampleActionFilterImpl(ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<SampleActionFilterAttribute>();
        }

        public void OnActionExecuting(ActionExecutingContext context)
        {
            _logger.LogInformation("Business action starting...");
            // perform some business logic work

        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            // perform some business logic work
            _logger.LogInformation("Business action completed.");
        }
```

```
    }
}
```

This filter can be applied to classes or methods using the `[SampleActionFilter]` syntax, instead of having to use `[TypeFilter]` or `[ServiceFilter]`.

: Avoid creating and using filters purely for logging purposes, since the *built-in framework logging features* should already provide what you need for logging. If you're going to add logging to your filters, it should focus on business domain concerns or behavior specific to your filter, rather than MVC actions or other framework events.

`IFilterFactory` implements `IFilter`. Therefore, an `IFilterFactory` instance can be used as an `IFilter` instance anywhere in the filter pipeline. When the framework prepares to invoke the filter, attempts to cast it to an `IFilterFactory`. If that cast succeeds, the `CreateInstance` method is called to create the `IFilter` instance that will be invoked. This provides a very flexible design, since the precise filter pipeline does not need to be set explicitly when the application starts.

You can implement `IFilterFactory` on your own attribute implementations as another approach to creating filters:

```csharp
public class AddHeaderWithFactoryAttribute : Attribute, IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new InternalAddHeaderFilter();
    }

    private class InternalAddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "Internal", new string[] { "Header Added" });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }
}
```

Ordering Filters can be applied to action methods or controllers (via attribute) or added to the global filters collection. Scope also generally determines ordering. The filter closest to the action runs first; generally you get overriding behavior without having to explicitly set ordering. This is sometimes referred to as "Russian doll" nesting, as each increase in scope is wrapped around the previous scope, like a nesting doll.

In addition to scope, filters can override their sequence of execution by implementing `IOrderedFilter`. This interface simply exposes an `int Order` property, and filters execute in ascending numeric order based on this property. All of the built-in filters, including `TypeFilterAttribute` and `ServiceFilterAttribute`, implement `IOrderedFilter`, so you can specify the order of filters when you apply the attribute to a class or method. By default, the `Order` property is 0 for all of the built-in filters, so scope is used as a tie-breaker and (unless `Order` is set to a non-zero value) is the determining factor.

Every controller that inherits from the `Controller` base class includes `OnActionExecuting` and `OnActionExecuted` methods. These methods wrap the filters that run for a given action, running first and last. The scope-based order, assuming no `Order` has been set for any filter, is:

1. The Controller `OnActionExecuting`

---

2. The Global filter `OnActionExecuting`

3. The Class filter `OnActionExecuting`

4. The Method filter `OnActionExecuting`

5. The Method filter `OnActionExecuted`

6. The Class filter `OnActionExecuted`

7. The Global filter `OnActionExecuted`

8. The Controller `OnActionExecuted`

: `IOrderedFilter` trumps scope when determining the order in which filters will run. Filters are sorted first by order, then scope is used to break ties. Order defaults to 0 if not set.

To modify the default, scope-based order, you could explicitly set the `Order` property of a class-level or method-level filter. For example, adding `Order=-1` to a method level attribute:

```
[MyFilter(Name = "Method Level Attribute", Order=-1)]
```

In this case, a value of less than zero would ensure this filter ran before both the Global and Class level filters (assuming their `Order` property was not set).

The new order would be:

1. The Controller `OnActionExecuting`

2. The Method filter `OnActionExecuting`

3. The Global filter `OnActionExecuting`

4. The Class filter `OnActionExecuting`

5. The Class filter `OnActionExecuted`

6. The Global filter `OnActionExecuted`

7. The Method filter `OnActionExecuted`

8. The Controller `OnActionExecuted`

: The `Controller` class's methods always run before and after all filters. These methods are not implemented as `IFilter` instances and do not participate in the `IFilter` ordering algorithm.

Authorization Filters

Authorization Filters control access to action methods, and are the first filters to be executed within the filter pipeline. They have only a before stage, unlike most filters that support before and after methods. You should only write a custom authorization filter if you are writing your own authorization framework. Note that you should not throw exceptions within authorization filters, since nothing will handle the exception (exception filters won't handle them). Instead, issue a challenge or find another way.

Learn more about *Authorization*.

Resource Filters

Resource Filters implement either the `IResourceFilter` or `IAsyncResourceFilter` interface, and their execution wraps most of the filter pipeline (only *Authorization Filters* run before them - all other filters and action processing happens between their `OnResourceExecuting` and `OnResourceExecuted` methods). Resource filters are especially useful if you need to short-circuit most of the work a request is doing. Caching would be one example use case for a resource filter, since if the response is already in the cache, the filter can immediately set a result and avoid the rest of the processing for the action.

The *short circuiting resource filter* shown above is one example of a resource filter. A very naive cache implementation (do not use this in production) that only works with `ContentResult` action results is shown below:

```csharp
public class NaiveCacheResourceFilterAttribute : Attribute,
    IResourceFilter
{
    private static readonly Dictionary<string, object> _cache
                = new Dictionary<string, object>();
    private string _cacheKey;

    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        _cacheKey = context.HttpContext.Request.Path.ToString();
        if (_cache.ContainsKey(_cacheKey))
        {
            var cachedValue = _cache[_cacheKey] as string;
            if (cachedValue != null)
            {
                context.Result = new ContentResult()
                { Content = cachedValue };
            }
        }
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
        if (!String.IsNullOrEmpty(_cacheKey) &&
            !_cache.ContainsKey(_cacheKey))
        {
            var result = context.Result as ContentResult;
            if (result != null)
            {
                _cache.Add(_cacheKey, result.Content);
            }
        }
    }
}
```

In `OnResourceExecuting`, if the result is already in the static dictionary cache, the `Result` property is set on `context`, and the action short-circuits and returns with the cached result. In the `OnResourceExecuted` method, if the current request's key isn't already in use, the current `Result` is stored in the cache, to be used by future requests.

Adding this filter to a class or method is shown here:

```csharp
[TypeFilter(typeof(NaiveCacheResourceFilterAttribute))]
public class CachedController : Controller
{
    public IActionResult Index()
    {
        return Content("This content was generated at " + DateTime.Now);
```

```
    }
}
```

Action Filters

Action Filters implement either the `IActionFilter` or `IAsyncActionFilter` interface and their execution surrounds the execution of action methods. Action filters are ideal for any logic that needs to see the results of model binding, or modify the controller or inputs to an action method. Additionally, action filters can view and directly modify the result of an action method.

The `OnActionExecuting` method runs before the action method, so it can manipulate the inputs to the action by changing `ActionExecutingContext.ActionArguments` or manipulate the controller through `ActionExecutingContext.Controller`. An `OnActionExecuting` method can short-circuit execution of the action method and subsequent action filters by setting `ActionExecutingContext.Result`. Throwing an exception in an `OnActionExecuting` method will also prevent execution of the action method and subsequent filters, but will be treated as a failure instead of successful result.

The `OnActionExecuted` method runs after the action method and can see and manipulate the results of the action through the `ActionExecutedContext.Result` property. `ActionExecutedContext.Canceled` will be set to true if the action execution was short-circuited by another filter. `ActionExecutedContext.Exception` will be set to a non-null value if the action or a subsequent action filter threw an exception. Setting `ActionExecutedContext.Exception` to null effectively 'handles' an exception, and `ActionExectedContext.Result` will then be executed as if it were returned from the action method normally.

For an `IAsyncActionFilter` the `OnActionExecutionAsync` combines all the possibilities of `OnActionExecuting` and `OnActionExecuted`. A call to `await next()` on the `ActionExecutionDelegate` will execute any subsequent action filters and the action method, returning an `ActionExecutedContext`. To short-circuit inside of an `OnActionExecutionAsync`, assign `ActionExecutingContext.Result` to some result instance and do not call the `ActionExectionDelegate`.

Exception Filters

Exception Filters implement either the `IExceptionFilter` or `IAsyncExceptionFilter` interface.

Exception filters handle unhandled exceptions, including those that occur during controller creation and *model binding*. They are only called when an exception occurs in the pipeline. They can provide a single location to implement common error handling policies within an app. The framework provides an abstract `ExceptionFilterAttribute` that you should be able to subclass for your needs. Exception filters are good for trapping exceptions that occur within MVC actions, but they're not as flexible as error handling middleware. Prefer middleware for the general case, and use filters only where you need to do error handling *differently* based on which MVC action was chosen.

: One example where you might need a different form of error handling for different actions would be in an app that exposes both API endpoints and actions that return views/HTML. The API endpoints could return error information as JSON, while the view-based actions could return an error page as HTML.

Exception filters do not have two events (for before and after) - they only implement `OnException` (or `OnExceptionAsync`). The `ExceptionContext` provided in the `OnException` parameter includes the `Exception` that occurred. If you set `context.ExceptionHandled` to `true`, the effect is that you've handled the exception, so the request will proceed as if it hadn't occurred (generally returning a 200 OK status). The following filter uses a custom developer error view to display details about exceptions that occur when the application is in development:

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;

namespace FiltersSample.Filters
{
    public class CustomExceptionFilterAttribute : ExceptionFilterAttribute
    {
        private readonly IHostingEnvironment _hostingEnvironment;
        private readonly IModelMetadataProvider _modelMetadataProvider;

        public CustomExceptionFilterAttribute(
            IHostingEnvironment hostingEnvironment,
            IModelMetadataProvider modelMetadataProvider)
        {
            _hostingEnvironment = hostingEnvironment;
            _modelMetadataProvider = modelMetadataProvider;
        }

        public override void OnException(ExceptionContext context)
        {
            if (!_hostingEnvironment.IsDevelopment())
            {
                // do nothing
                return;
            }
            var result = new ViewResult {ViewName = "CustomError"};
            result.ViewData = new ViewDataDictionary(_modelMetadataProvider,context.ModelState);
            result.ViewData.Add("Exception", context.Exception);
            // TODO: Pass additional detailed data via ViewData
            context.ExceptionHandled = true; // mark exception as handled
            context.Result = result;
        }
    }
}
```

Result Filters

Result Filters implement either the IResultFilter or IAsyncResultFilter interface and their execution surrounds the execution of action results. Result filters are only executed for successful results - when the action or action filters produce an action result. Result filters are not executed when exception filters handle an exception, unless the exception filter sets Exception = null.

: The kind of result being executed depends on the action in question. An MVC action returning a view would include all razor processing as part of the ViewResult being executed. An API method might perform some serialization as part of the execution of the result. Learn more about *action results*

Result filters are ideal for any logic that needs to directly surround view execution or formatter execution. Result filters can replace or modify the action result that's responsible for producing the response.

The OnResultExecuting method runs before the action result is executed, so it can manipulate the action result through ResultExecutingContext.Result. An OnResultExecuting method can short-circuit execution of the action result and subsequent result filters by setting ResultExecutingContext.Cancel to true. If short-

circuited, MVC will not modify the response; you should generally write to the response object directly when short-circuiting to avoid generating an empty response. Throwing an exception in an `OnResultExecuting` method will also prevent execution of the action result and subsequent filters, but will be treated as a failure instead of a successful result.

The `OnResultExecuted` method runs after the action result has executed. At this point if no exception was thrown, the response has likely been sent to the client and cannot be changed further. `ResultExecutedContext.Canceled` will be set to true if the action result execution was short-circuited by another filter. `ResultExecutedContext.Exception` will be set to a non-null value if the action result or a subsequent result filter threw an exception. Setting `ResultExecutedContext.Exception` to null effectively 'handles' an exception and will prevent the exeception from being rethrown by MVC later in the pipeline. If handling an exception in a result filter, consider whether or not it's appropriate to write any data to the response. If the action result throws partway through its execution, and the headers have already been flushed to the client, there's no reliable mechanism to send a failure code.

For an `IAsyncResultFilter` the `OnResultExecutionAsync` combines all the possibilities of `OnResultExecuting` and `OnResultExecuted`. A call to `await next()` on the `ResultExecutionDelegate` will execute any subsequent result filters and the action result, returning a `ResultExecutedContext`. To short-circuit inside of an `OnResultExecutionAsync`, set `ResultExecutingContext.Cancel` to true and do not call the `ResultExectionDelegate`.

You can override the built-in `ResultFilterAttribute` to create result filters. The *AddHeaderAttribute* class shown above is an example of a result filter.

: If you need to add headers to the response, do so before the action result executes. Otherwise, the response may have been sent to the client, and it will be too late to modify it. For a result filter, this means adding the header in `OnResultExecuting` rather than `OnResultExecuted`.

Filters vs. Middleware

In general, filters are meant to handle cross-cutting business and application concerns. This is often the same use case for *middleware*. Filters are very similar to middleware in capability, but let you scope that behavior and insert it into a location in your app where it makes sense, such as before a view, or after model binding. Filters are a part of MVC, and have access to its context and constructs. For instance, middleware can't easily detect whether model validation on a request has generated errors, and respond accordingly, but a filter can easily do so.

To experiment with filters, download, test and modify the sample.

Dependency Injection and Controllers

By Steve Smith

ASP.NET Core MVC controllers should request their dependencies explicitly via their constructors. In some instances, individual controller actions may require a service, and it may not make sense to request at the controller level. In this case, you can also choose to inject a service as a parameter on the action method.

Sections:

- *Dependency Injection*
- *Constructor Injection*
- *Action Injection with FromServices*
- *Accessing Settings from a Controller*

View or download sample code

Dependency Injection

Dependency injection is a technique that follows the Dependency Inversion Principle, allowing for applications to be composed of loosely coupled modules. ASP.NET Core has built-in support for *dependency injection*, which makes applications easier to test and maintain.

Constructor Injection

ASP.NET Core's built-in support for constructor-based dependency injection extends to MVC controllers. By simply adding a service type to your controller as a constructor parameter, ASP.NET will attempt to resolve that type using its built in service container. Services are typically, but not always, defined using interfaces. For example, if your application has business logic that depends on the current time, you can inject a service that retrieves the time (rather than hard-coding it), which would allow your tests to pass in implementations that use a set time.

```
using System;

namespace ControllerDI.Interfaces
{
    public interface IDateTime
    {
        DateTime Now { get; }
    }
}
```

Implementing an interface like this one so that it uses the system clock at runtime is trivial:

```
using System;
using ControllerDI.Interfaces;

namespace ControllerDI.Services
{
    public class SystemDateTime : IDateTime
    {
        public DateTime Now
        {
            get { return DateTime.Now; }
        }
    }
}
```

With this in place, we can use the service in our controller. In this case, we have added some logic to the `HomeController` Index method to display a greeting to the user based on the time of day.

```
using ControllerDI.Interfaces;
using Microsoft.AspNet.Mvc;

namespace ControllerDI.Controllers
{
    public class HomeController : Controller
    {
        private readonly IDateTime _dateTime;

        public HomeController(IDateTime dateTime)
        {
```

```
12          _dateTime = dateTime;
13      }
14
15      public IActionResult Index()
16      {
17          var serverTime = _dateTime.Now;
18          if (serverTime.Hour < 12)
19          {
20              ViewData["Message"] = "It's morning here – Good Morning!";
21          }
22          else if (serverTime.Hour < 17)
23          {
24              ViewData["Message"] = "It's afternoon here – Good Afternoon!";
25          }
26          else
27          {
28              ViewData["Message"] = "It's evening here – Good Evening!";
29          }
30          return View();
31      }
32  }
33 }
```

If we run the application now, we will most likely encounter an error:

```
An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'ControllerDI.Interfaces.IDateTime' whi
Microsoft.Extensions.DependencyInjection.ActivatorUtilities.GetService(IServiceProvider sp, Type type
```

This error occurs when we have not configured a service in the ConfigureServices method in our Startup class. To specify that requests for IDateTime should be resolved using an instance of SystemDateTime, add the highlighted line in the listing below to your ConfigureServices method:

```
1  public void ConfigureServices(IServiceCollection services)
2  {
3      services.AddMvc();
4
5      // Add application services.
6      services.AddTransient<IDateTime, SystemDateTime>();
7  }
```

: This particular service could be implemented using any of several different lifetime options (Transient, Scoped, or Singleton). See *Dependency Injection* to understand how each of these scope options will affect the behavior of your service.

Once the service has been configured, running the application and navigating to the home page should display the time-based message as expected:

A Message From The Server

It's afternoon here - Good Afternoon!

: See *Testing Controller Logic* to learn how to explicitly request dependencies http://deviq.com/
explicit-dependencies-principle in controllers makes code easier to test.

ASP.NET Core's built-in dependency injection supports having only a single constructor for classes requesting services. If you have more than one constructor, you may get an exception stating:

```
An unhandled exception occurred while processing the request.

InvalidOperationException: Multiple constructors accepting all given argument types have been found :
Microsoft.Extensions.DependencyInjection.ActivatorUtilities.FindApplicableConstructor(Type instanceTy
```

As the error message states, you can correct this problem having just a single constructor. You can also *replace the default dependency injection support with a third party implementation*, many of which support multiple constructors.

Action Injection with FromServices

Sometimes you don't need a service for more than one action within your controller. In this case, it may make sense to inject the service as a parameter to the action method. This is done by marking the parameter with the attribute `[FromServices]` as shown here:

```csharp
public IActionResult About([FromServices] IDateTime dateTime)
{
    ViewData["Message"] = "Currently on the server the time is " + dateTime.Now;

    return View();
}
```

Accessing Settings from a Controller

Accessing application or configuration settings from within a controller is a common pattern. This access should use the Options pattern described in *configuration*. You generally should not request settings directly from your controller using dependency injection. A better approach is to request an `IOptions<T>` instance, where `T` is the configuration class you need.

To work with the options pattern, you need to create a class that represents the options, such as this one:

```csharp
namespace ControllerDI.Model
{
    public class SampleWebSettings
    {
        public string Title { get; set; }
        public int Updates { get; set; }
```

```
7        }
8  }
```

Then you need to configure the application to use the options model and add your configuration class to the services collection in `ConfigureServices`:

```
1  public Startup()
2  {
3      var builder = new ConfigurationBuilder()
4          .AddJsonFile("samplewebsettings.json");
5      Configuration = builder.Build();
6  }
7
8  public IConfigurationRoot Configuration { get; set; }
9
10 // This method gets called by the runtime. Use this method to add services to the container.
11 // For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?
12 public void ConfigureServices(IServiceCollection services)
13 {
14     // Required to use the Options<T> pattern
15     services.AddOptions();
16
17     // Add settings from configuration
18     services.Configure<SampleWebSettings>(Configuration);
19
20     // Uncomment to add settings from code
21     //services.Configure<SampleWebSettings>(settings =>
22     //{
23     //    settings.Updates = 17;
24     //});
25
26     services.AddMvc();
27
28     // Add application services.
29     services.AddTransient<IDateTime, SystemDateTime>();
30 }
```

: In the above listing, we are configuring the application to read the settings from a JSON-formatted file. You can also configure the settings entirely in code, as is shown in the commented code above. See *Configuration* for further configuration options.

Once you've specified a strongly-typed configuration object (in this case, `SampleWebSettings`) and added it to the services collection, you can request it from any Controller or Action method by requesting an instance of `IOptions<T>` (in this case, `IOptions<SampleWebSettings>`). The following code shows how one would request the settings from a controller:

```
1      public class SettingsController : Controller
2      {
3          private readonly SampleWebSettings _settings;
4
5          public SettingsController(IOptions<SampleWebSettings> settingsOptions )
6          {
7              _settings = settingsOptions.Value;
8          }
9
10         public IActionResult Index()
```

```
11          {
12              ViewData["Title"] = _settings.Title;
13              ViewData["Updates"] = _settings.Updates;
14              return View();
15          }
16      }
```

Following the Options pattern allows settings and configuration to be decoupled from one another, and ensures the controller is following separation of concerns, since it doesn't need to know how or where to find the settings information. It also makes the controller easier to unit test *Testing Controller Logic*, since there is no static cling or direct instantiation of settings classes within the controller class.

Testing Controller Logic

By Steve Smith

Controllers in ASP.NET MVC apps should be small and focused on user-interface concerns. Large controllers that deal with non-UI concerns are more difficult to test and maintain.

Sections

- *Why Test Controllers*
- *Unit Testing*
- *Integration Testing*

View or download sample from GitHub

Why Test Controllers

Controllers are a central part of any ASP.NET Core MVC application. As such, you should have confidence they behave as intended for your app. Automated tests can provide you with this confidence and can detect errors before they reach production. It's important to avoid placing unnecessary responsibilities within your controllers and ensure your tests focus only on controller responsibilities.

Controller logic should be minimal and not be focused on business logic or infrastructure concerns (for example, data access). Test controller logic, not the framework. Test how the controller *behaves* based on valid or invalid inputs. Test controller responses based on the result of the business operation it performs.

Typical controller responsibilities:

- Verify `ModelState.IsValid`

- Return an error response if `ModelState` is invalid

- Retrieve a business entity from persistence

- Perform an action on the business entity

- Save the business entity to persistence

- Return an appropriate `IActionResult`

Unit Testing

Unit testing involves testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action is tested, not the behavior of its dependencies or of the framework

itself. As you unit test your controller actions, make sure you focus only on its behavior. A controller unit test avoids things like *filters*, *routing*, or *model binding*. By focusing on testing just one thing, unit tests are generally simple to write and quick to run. A well-written set of unit tests can be run frequently without much overhead. However, unit tests do not detect issues in the interaction between components, which is the purpose of *integration testing*.

If you've writing custom filters, routes, etc, you should unit test them, but not as part of your tests on a particular controller action. They should be tested in isolation.

: Create and run unit tests with Visual Studio.

To demonstrate unit testing, review the following controller. It displays a list of brainstorming sessions and allows new brainstorming sessions to be created with a POST:

```csharp
using System;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;

namespace TestingControllersSample.Controllers
{
    public class HomeController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public HomeController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        public async Task<IActionResult> Index()
        {
            var sessionList = await _sessionRepository.ListAsync();

            var model = sessionList.Select(session => new StormSessionViewModel()
            {
                Id = session.Id,
                DateCreated = session.DateCreated,
                Name = session.Name,
                IdeaCount = session.Ideas.Count
            });

            return View(model);
        }

        public class NewSessionModel
        {
            [Required]
            public string SessionName { get; set; }
        }

        [HttpPost]
        public async Task<IActionResult> Index(NewSessionModel model)
        {
```

```
                if (!ModelState.IsValid)
                {
                    return BadRequest(ModelState);
                }

                await _sessionRepository.AddAsync(new BrainstormSession()
                {
                    DateCreated = DateTimeOffset.Now,
                    Name = model.SessionName
                });

                return RedirectToAction("Index");
            }
        }
    }
```

The controller is following the explicit dependencies principle, expecting dependency injection to provide it with an instance of IBrainstormSessionRepository. This makes it fairly easy to test using a mock object framework, like Moq. The HTTP GET Index method has no looping or branching and only calls one method. To test this Index method, we need to verify that a ViewResult is returned, with a ViewModel from the repository's List method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Controllers;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class HomeControllerTests
    {
        [Fact]
        public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
        {
            // Arrange
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
            var controller = new HomeController(mockRepo.Object);

            // Act
            var result = await controller.Index();

            // Assert
            var viewResult = Assert.IsType<ViewResult>(result);
            var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
                viewResult.ViewData.Model);
            Assert.Equal(2, model.Count());
        }

        [Fact]
        public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
```

```
        {
            // Arrange
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
            var controller = new HomeController(mockRepo.Object);
            controller.ModelState.AddModelError("SessionName", "Required");
            var newSession = new HomeController.NewSessionModel();

            // Act
            var result = await controller.Index(newSession);

            // Assert
            var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
            Assert.IsType<SerializableError>(badRequestResult.Value);
        }

        [Fact]
        public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
        {
            // Arrange
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
                .Returns(Task.CompletedTask)
                .Verifiable();
            var controller = new HomeController(mockRepo.Object);
            var newSession = new HomeController.NewSessionModel()
            {
                SessionName = "Test Name"
            };

            // Act
            var result = await controller.Index(newSession);

            // Assert
            var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
            Assert.Null(redirectToActionResult.ControllerName);
            Assert.Equal("Index", redirectToActionResult.ActionName);
            mockRepo.Verify();
        }

        private List<BrainstormSession> GetTestSessions()
        {
            var sessions = new List<BrainstormSession>();
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 2),
                Id = 1,
                Name = "Test One"
            });
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 1),
                Id = 2,
                Name = "Test Two"
            });
            return sessions;
        }
    }
```

```
}
```

The `HTTP POST Index` method (shown below) should verify:

- The action method returns a `ViewResult` with the appropriate data when `ModelState.IsValid` is `false`

- The `Add` method on the repository is called and a `RedirectToActionResult` is returned with the correct arguments when `ModelState.IsValid` is true.

```csharp
        var viewResult = Assert.IsType<ViewResult>(result);
        var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
            viewResult.ViewData.Model);
        Assert.Equal(2, model.Count());
    }

    [Fact]
    public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
    {
        // Arrange
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
        var controller = new HomeController(mockRepo.Object);
        controller.ModelState.AddModelError("SessionName", "Required");
        var newSession = new HomeController.NewSessionModel();

        // Act
        var result = await controller.Index(newSession);

        // Assert
        var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
        Assert.IsType<SerializableError>(badRequestResult.Value);
    }

    [Fact]
    public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
    {
        // Arrange
        var mockRepo = new Mock<IBrainstormSessionRepository>();
```

The first test confirms when `ModelState` is not valid, the same `ViewResult` is returned as for a `GET` request. Note that the test doesn't attempt to pass in an invalid model. That wouldn't work anyway since model binding isn't running - we're just calling the method directly. However, we're not trying to test model binding - we're only testing what our code in the action method does. The simplest approach is to add an error to `ModelState`.

The second test verifies that when `ModelState` is valid, a new `BrainstormSession` is added (via the repository), and the method returns a `RedirectToActionResult` with the expected properties. Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows it to be verified in the test. This is done with the call to `mockRepo.Verify`.

: The Moq library used in this sample makes it easy to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

Another controller in the app displays information related to a particular brainstorming session. It includes some logic to deal with invalid id values:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.ViewModels;

namespace TestingControllersSample.Controllers
{
    public class SessionController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public SessionController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        public async Task<IActionResult> Index(int? id)
        {
            if (!id.HasValue)
            {
                return RedirectToAction("Index", "Home");
            }

            var session = await _sessionRepository.GetByIdAsync(id.Value);
            if (session == null)
            {
                return Content("Session not found.");
            }

            var viewModel = new StormSessionViewModel()
            {
                DateCreated = session.DateCreated,
                Name = session.Name,
                Id = session.Id
            };

            return View(viewModel);
        }
    }
}
```

The controller action has three cases to test, one for each `return` statement:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Controllers;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class SessionControllerTests
```

```
    {
        [Fact]
        public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
        {
            // Arrange
            var controller = new SessionController(sessionRepository: null);

            // Act
            var result = await controller.Index(id: null);

            // Arrange
            var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
            Assert.Equal("Home", redirectToActionResult.ControllerName);
            Assert.Equal("Index", redirectToActionResult.ActionName);
        }


        [Fact]
        public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
        {
            // Arrange
            int testSessionId = 1;
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
                .Returns(Task.FromResult((BrainstormSession)null));
            var controller = new SessionController(mockRepo.Object);

            // Act
            var result = await controller.Index(testSessionId);

            // Assert
            var contentResult = Assert.IsType<ContentResult>(result);
            Assert.Equal("Session not found.", contentResult.Content);
        }


        [Fact]
        public async Task IndexReturnsViewResultWithStormSessionViewModel()
        {
            // Arrange
            int testSessionId = 1;
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
                .Returns(Task.FromResult(GetTestSessions().FirstOrDefault(s => s.Id == testSessionId)
            var controller = new SessionController(mockRepo.Object);

            // Act
            var result = await controller.Index(testSessionId);

            // Assert
            var viewResult = Assert.IsType<ViewResult>(result);
            var model = Assert.IsType<StormSessionViewModel>(viewResult.ViewData.Model);
            Assert.Equal("Test One", model.Name);
            Assert.Equal(2, model.DateCreated.Day);
            Assert.Equal(testSessionId, model.Id);
        }

        private List<BrainstormSession> GetTestSessions()
        {
            var sessions = new List<BrainstormSession>();
```

```
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 2),
                Id = 1,
                Name = "Test One"
            });
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 1),
                Id = 2,
                Name = "Test Two"
            });
            return sessions;
        }
    }
}
```

The app exposes functionality as a web API (a list of ideas associated with a brainstorming session and a method for adding new ideas to a session):

```csharp
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;

namespace TestingControllersSample.Api
{
    [Route("api/ideas")]
    public class IdeasController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public IdeasController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        [Route("forsession/{sessionId}")]
        [HttpGet]
        public async Task<IActionResult> ForSession(int sessionId)
        {
            var session = await _sessionRepository.GetByIdAsync(sessionId);
            if (session == null)
            {
                return NotFound(sessionId);
            }

            var result = session.Ideas.Select(idea => new IdeaDTO()
            {
                Id = idea.Id,
                Name = idea.Name,
                Description = idea.Description,
                DateCreated = idea.DateCreated
            }).ToList();
```

```
            return Ok(result);
        }

        [Route("create")]
        [HttpPost]
        public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            var session = await _sessionRepository.GetByIdAsync(model.SessionId);
            if (session == null)
            {
                return NotFound(model.SessionId);
            }

            var idea = new Idea()
            {
                DateCreated = DateTimeOffset.Now,
                Description = model.Description,
                Name = model.Name
            };
            session.AddIdea(idea);

            await _sessionRepository.UpdateAsync(session);

            return Ok(session);
        }
    }
}
```

The `ForSession` method returns a list of `IdeaDTO` types, with property names camel cased to match JavaScript conventions. Avoid returning your business domain entities directly via API calls, since frequently they include more data than the API client requires, and they unnecessarily couple your app's internal domain model with the API you expose externally. Mapping between domain entities and the types you will return over the wire can be done manually (using a LINQ `Select` as shown here) or using a library like AutoMapper

The unit tests for the `Create` and `ForSession` API methods:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Api;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class ApiIdeasControllerTests
    {
        [Fact]
```

```csharp
        public async Task Create_ReturnsBadRequest_GivenInvalidModel()
        {
            // Arrange & Act
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            var controller = new IdeasController(mockRepo.Object);
            controller.ModelState.AddModelError("error","some error");

            // Act
            var result = await controller.Create(model: null);

            // Assert
            Assert.IsType<BadRequestObjectResult>(result);
        }

        [Fact]
        public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
        {
            // Arrange
            int testSessionId = 123;
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
                .Returns(Task.FromResult((BrainstormSession)null));
            var controller = new IdeasController(mockRepo.Object);

            // Act
            var result = await controller.Create(new NewIdeaModel());

            // Assert
            Assert.IsType<NotFoundObjectResult>(result);
        }

        [Fact]
        public async Task Create_ReturnsNewlyCreatedIdeaForSession()
        {
            // Arrange
            int testSessionId = 123;
            string testName = "test name";
            string testDescription = "test description";
            var testSession = GetTestSession();
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
                .Returns(Task.FromResult(testSession));
            var controller = new IdeasController(mockRepo.Object);

            var newIdea = new NewIdeaModel()
            {
                Description = testDescription,
                Name = testName,
                SessionId = testSessionId
            };
            mockRepo.Setup(repo => repo.UpdateAsync(testSession))
                .Returns(Task.CompletedTask)
                .Verifiable();

            // Act
            var result = await controller.Create(newIdea);

            // Assert
```

```csharp
        var okResult = Assert.IsType<OkObjectResult>(result);
        var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
        mockRepo.Verify();
        Assert.Equal(2, returnSession.Ideas.Count());
        Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
        Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
    }

    [Fact]
    public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
    {
        // Arrange
        int testSessionId = 123;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .Returns(Task.FromResult((BrainstormSession)null));
        var controller = new IdeasController(mockRepo.Object);

        // Act
        var result = await controller.ForSession(testSessionId);

        // Assert
        var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
        Assert.Equal(testSessionId, notFoundObjectResult.Value);
    }

    [Fact]
    public async Task ForSession_ReturnsIdeasForSession()
    {
        // Arrange
        int testSessionId = 123;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId)).Returns(Task.FromResult(GetTestS
        var controller = new IdeasController(mockRepo.Object);

        // Act
        var result = await controller.ForSession(testSessionId);

        // Assert
        var okResult = Assert.IsType<OkObjectResult>(result);
        var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
        var idea = returnValue.FirstOrDefault();
        Assert.Equal("One", idea.Name);
    }

    private BrainstormSession GetTestSession()
    {
        var session = new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 2),
            Id = 1,
            Name = "Test One"
        };

        var idea = new Idea() { Name = "One" };
        session.AddIdea(idea);
        return session;
    }
```

```
    }
}
```

As stated previously, to test the behavior of the method when `ModelState` is invalid, add a model error to the controller as part of the test. Don't try to test model validation or model binding in your unit tests - just test your action method's behavior when confronted with a particular `ModelState` value.

The second test depends on the repository returning null, so the mock repository is configured to return null. There's no need to create a test database (in memory or otherwise) and construct a query that will return this result - it can be done in a single line as shown.

The last test verifies that the repository's `Update` method is called. As we did previously, the mock is called with `Verifiable` and then the mocked repository's `Verify` method is called to confirm the verifiable method was executed. It's not a unit test responsibility to ensure that the `Update` method saved the data; that can be done with an integration test.

Integration Testing

Integration testing is done to ensure separate modules within your app work correctly together. Generally, anything you can test with a unit test, you can also test with an integration test, but the reverse isn't true. However, integration tests tend to be much slower than unit tests. Thus, it's best to test whatever you can with unit tests, and use integration tests for scenarios that involve multiple collaborators.

Although they may still be useful, mock objects are rarely used in integration tests. In unit testing, mock objects are an effective way to control how collaborators outside of the unit being tested should behave for the purposes of the test. In an integration test, real collaborators are used to confirm the whole subsystem works together correctly.

Application State   One important consideration when performing integration testing is how to set your app's state. Tests need to run independent of one another, and so each test should start with the app in a known state. If your app doesn't use a database or have any persistence, this may not be an issue. However, most real-world apps persist their state to some kind of data store, so any modifications made by one test could impact another test unless the data store is reset. Using the built-in `TestServer`, it's very straightforward to host ASP.NET Core apps within our integration tests, but that doesn't necessarily grant access to the data it will use. If you're using an actual database, one approach is to have the app connect to a test database, which your tests can access and ensure is reset to a known state before each test executes.

In this sample application, I'm using Entity Framework Core's InMemoryDatabase support, so I can't just connect to it from my test project. Instead, I expose an `InitializeDatabase` method from the app's `Startup` class, which I call when the app starts up if it's in the `Development` environment. My integration tests automatically benefit from this as long as they set the environment to `Development`. I don't have to worry about resetting the database, since the InMemoryDatabase is reset each time the app restarts.

The `Startup` class:

```csharp
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.Infrastructure;
```

```
namespace TestingControllersSample
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(
                optionsBuilder => optionsBuilder.UseInMemoryDatabase());

            services.AddMvc();

            services.AddScoped<IBrainstormSessionRepository,
                EFStormSessionRepository>();
        }

        public void Configure(IApplicationBuilder app,
            IHostingEnvironment env,
            ILoggerFactory loggerFactory)
        {
            loggerFactory.AddConsole(LogLevel.Warning);

            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();

                var repository = app.ApplicationServices.GetService<IBrainstormSessionRepository>();
                InitializeDatabaseAsync(repository).Wait();
            }

            app.UseStaticFiles();

            app.UseMvcWithDefaultRoute();
        }

        public async Task InitializeDatabaseAsync(IBrainstormSessionRepository repo)
        {
            var sessionList = await repo.ListAsync();
            if (!sessionList.Any())
            {
                await repo.AddAsync(GetTestSession());
            }
        }

        public static BrainstormSession GetTestSession()
        {
            var session = new BrainstormSession()
            {
                Name = "Test Session 1",
                DateCreated = new DateTime(2016, 8, 1)
            };
            var idea = new Idea()
            {
                DateCreated = new DateTime(2016, 8, 1),
                Description = "Totally awesome idea",
                Name = "Awesome idea"
            };
            session.AddIdea(idea);
            return session;
```

```
        }
    }
}
```

You'll see the `GetTestSession` method used frequently in the integration tests below.

Accessing Views   Each integration test class configures the `TestServer` that will run the ASP.NET Core app. By default, `TestServer` hosts the web app in the folder where it's running - in this case, the test project folder. Thus, when you attempt to test controller actions that return `ViewResult`, you may see this error:

```
The view 'Index' was not found. The following locations were searched:
(list of locations)
```

To correct this issue, you need to configure the server to use the `ApplicationBasePath` and `ApplicationName` of the web project. This is done in the call to `UseServices` in the integration test class shown:

```csharp
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Xunit;

namespace TestingControllersSample.Tests.IntegrationTests
{
    public class HomeControllerTests : IClassFixture<TestFixture<TestingControllersSample.Startup>>
    {
        private readonly HttpClient _client;

        public HomeControllerTests(TestFixture<TestingControllersSample.Startup> fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task ReturnsInitialListOfBrainstormSessions()
        {
            // Arrange
            var testSession = Startup.GetTestSession();

            // Act
            var response = await _client.GetAsync("/");

            // Assert
            response.EnsureSuccessStatusCode();
            var responseString = await response.Content.ReadAsStringAsync();
            Assert.True(responseString.Contains(testSession.Name));
        }

        [Fact]
        public async Task PostAddsNewBrainstormSession()
        {
            // Arrange
            string testSessionName = Guid.NewGuid().ToString();
            var data = new Dictionary<string, string>();
            data.Add("SessionName", testSessionName);
            var content = new FormUrlEncodedContent(data);
```

```
        // Act
        var response = await _client.PostAsync("/", content);

        // Assert
        Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
        Assert.Equal("/", response.Headers.Location.ToString());
    }
  }
}
```

In the test above, the `responseString` gets the actual rendered HTML from the View, which can be inspected to confirm it contains expected results.

API Methods If your app exposes web APIs, it's a good idea to have automated tests confirm they execute as expected. The built-in `TestServer` makes it easy to test web APIs. If your API methods are using model binding, you should always check `ModelState.IsValid`, and integration tests are the right place to confirm that your model validation is working properly.

The following set of tests target the `Create` method in the *IdeasController* class shown above:

```
        [Fact]
        public async Task CreatePostReturnsBadRequestForMissingNameValue()
        {
            // Arrange
            var newIdea = new NewIdeaDto("", "Description", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForMissingDescriptionValue()
        {
            // Arrange
            var newIdea = new NewIdeaDto("Name", "", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForSessionIdValueTooSmall()
        {
            // Arrange
            var newIdea = new NewIdeaDto("Name", "Description", 0);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
```

```
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForSessionIdValueTooLarge()
        {
            // Arrange
            var newIdea = new NewIdeaDto("Name", "Description", 1000001);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }

        [Fact]
        public async Task CreatePostReturnsNotFoundForInvalidSession()
        {
            // Arrange
            var newIdea = new NewIdeaDto("Name", "Description", 123);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
        }

        [Fact]
        public async Task CreatePostReturnsCreatedIdeaWithCorrectInputs()
        {
            // Arrange
            var testIdeaName = Guid.NewGuid().ToString();
            var newIdea = new NewIdeaDto(testIdeaName, "Description", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            response.EnsureSuccessStatusCode();
            var returnedSession = await response.Content.ReadAsJsonAsync<BrainstormSession>();
            Assert.Equal(2, returnedSession.Ideas.Count);
            Assert.True(returnedSession.Ideas.Any(i => i.Name == testIdeaName));
        }

        [Fact]
        public async Task ForSessionReturnsNotFoundForBadSessionId()
        {
            // Arrange & Act
            var response = await _client.GetAsync("/api/ideas/forsession/500");

            // Assert
            Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
        }

        [Fact]
        public async Task ForSessionReturnsIdeasForValidSessionId()
        {
```

```
            // Arrange
            var testSession = Startup.GetTestSession();

            // Act
            var response = await _client.GetAsync("/api/ideas/forsession/1");

            // Assert
            response.EnsureSuccessStatusCode();
            var ideaList = JsonConvert.DeserializeObject<List<IdeaDTO>>(
                await response.Content.ReadAsStringAsync());
            var firstIdea = ideaList.First();
```

Unlike integration tests of actions that returns HTML views, web API methods that return results can usually be deserialized as strongly typed objects, as the last test above shows. In this case, the test deserializes the result to a `BrainstormSession` instance, and confirms that the idea was correctly added to its collection of ideas.

You'll find additional examples of integration tests in this article's sample project.

Areas

By *Tom Archer*

Areas provide a way to separate a large MVC application into semantically-related groups of models, views, and controllers. Let's take a look at an example to illustrate how Areas are created and used. Let's say you have a store app that has two distinct groupings of controllers and views: Products and Services.

Instead of having all of the controllers located under the Controllers parent directory, and all the views located under the Views parent directory, you could use Areas to group your views and controllers according to the area (or logical grouping) with which they're associated.

- Project name
  - Areas
    * Products
      · Controllers
      · HomeController.cs
      · Views
      · Home
      · Index.cshtml
    * Services
      · Controllers
      · HomeController.cs
      · Views
      · Home
      · Index.cshtml

Looking at the preceding directory hierarchy example, there are a few guidelines to keep in mind when defining areas:

- A directory called *Areas* must exist as a child directory of the project.
- The *Areas* directory contains a subdirectory for each of your project's areas (*Products* and *Services*, in this example).

- Your controllers should be located as follows: `/Areas/[area]/Controllers/[controller].cs`

- Your views should be located as follows: `/Areas/[area]/Views/[controller]/[action].cshtml`

Note that if you have a view that is shared across controllers, it can be located in either of the following locations:

- `/Areas/[area]/Views/Shared/[action].cshtml`

- `/Views/Shared/[action].cshtml`

Once you've defined the folder hierarchy, you need to tell MVC that each controller is associated with an area. You do that by decorating the controller name with the `[Area]` attribute.

```
...
namespace MyStore.Areas.Products.Controllers
{
    [Area("Products")]
    public class HomeController : Controller
    {
        // GET: /<controller>/
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

The final step is to set up a route definition that works with your newly created areas. The *Routing to Controller Actions* article goes into detail about how to create route definitions, including using conventional routes versus attribute routes. In this example, we'll use a conventional route. To do so, simply open the *Startup.cs* file and modify it by adding the highlighted route definition below.

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

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

Now, when the user browses to *http://<yourApp>/products*, the `Index` action method of the `HomeController` in the `Products` area will be invoked.

Linking between areas

To link between areas, you simply specify the area in which the controller is defined by using *Tag Helpers*.

The following snippet shows how to link to a controller action that is defined within an area named *Products*.

```
<a asp-area="Products" asp-controller="Home" asp-action="Index">See Products Home Page</a>
```

To link to a controller action that is not part of an area, simply remove `asp-route-area`

```
<a asp-controller="Home" asp-action="Index">Go to Home Page</a>
```

Summary

Areas are a very useful tool for grouping semantically-related controllers and actions under a common parent folder. In this article, you learned how to set up your folder hierarchy to support `Areas`, how to specify the `[Area]` attribute to denote a controller as belonging to a specified area, and how to define your routes with areas.

Working with the Application Model

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.6 Testing

1.6.1 Integration Testing

By Steve Smith

Integration testing ensures that an application's components function correctly when assembled together. ASP.NET Core supports integration testing using unit test frameworks and a built-in test web host that can be used to handle requests without network overhead.

Sections:

- *Introduction to Integration Testing*
- *Integration Testing ASP.NET Core*
- *Refactoring to use Middleware*
- *Summary*
- *Additional Resources*

View or download sample code

Introduction to Integration Testing

Integration tests verify that different parts of an application work correctly together. Unlike Unit testing, integration tests frequently involve application infrastructure concerns, such as a database, file system, network resources, or web requests and responses. Unit tests use fakes or mock objects in place of these concerns, but the purpose of integration tests is to confirm that the system works as expected with these systems.

Integration tests, because they exercise larger segments of code and because they rely on infrastructure elements, tend to be orders of magnitude slower than unit tests. Thus, it's a good idea to limit how many integration tests you write, especially if you can test the same behavior with a unit test.

: If some behavior can be tested using either a unit test or an integration test, prefer the unit test, since it will be almost always be faster. You might have dozens or hundreds of unit tests with many different inputs, but just a handful of integration tests covering the most important scenarios.

Testing the logic within your own methods is usually the domain of unit tests. Testing how your application works within its framework (e.g. ASP.NET Core) or with a database is where integration tests come into play. It doesn't take too many integration tests to confirm that you're able to write a row to and then read a row from the database. You don't need to test every possible permutation of your data access code - you only need to test enough to give you confidence that your application is working properly.

Integration Testing ASP.NET Core

To get set up to run integration tests, you'll need to create a test project, refer to your ASP.NET Core web project, and install a test runner. This process is described in the Unit testing documentation, along with more detailed instructions on running tests and recommendations for naming your tests and test classes.

: Separate your unit tests and your integration tests using different projects. This helps ensure you don't accidentally introduce infrastructure concerns into your unit tests, and lets you easily choose to run all tests, or just one set or the other.

The Test Host

ASP.NET Core includes a test host that can be added to integration test projects and used to host ASP.NET Core applications, serving test requests without the need for a real web host. The provided sample includes an integration test project which has been configured to use xUnit and the Test Host, as you can see from this excerpt from its *project.json* file:

```
"dependencies": {
  "PrimeWeb": "1.0.0",
  "xunit": "2.1.0",
  "dotnet-test-xunit": "1.0.0-rc2-build10025",
  "Microsoft.AspNetCore.TestHost": "1.0.0"
},
```

Once the Microsoft.AspNetCore.TestHost package is included in the project, you will be able to create and configure a TestServer in your tests. The following test shows how to verify that a request made to the root of a site returns "Hello World!" and should run successfully against the default ASP.NET Core Empty Web template created by Visual Studio.

```
private readonly TestServer _server;
private readonly HttpClient _client;
public PrimeWebDefaultRequestShould()
{
    // Arrange
    _server = new TestServer(new WebHostBuilder()
        .UseStartup<Startup>());
    _client = _server.CreateClient();
}

[Fact]
```

```
public async Task ReturnHelloWorld()
{
    // Act
    var response = await _client.GetAsync("/");
    response.EnsureSuccessStatusCode();

    var responseString = await response.Content.ReadAsStringAsync();

    // Assert
    Assert.Equal("Hello World!",
        responseString);
}
```

These tests are using the Arrange-Act-Assert pattern, but in this case all of the Arrange step is done in the constructor, which creates an instance of `TestServer`. A configured `WebHostBuilder` will be used to create a `TestHost`; in this example we are passing in the `Configure` method from our system under test (SUT)'s `Startup` class. This method will be used to configure the request pipeline of the `TestServer` identically to how the SUT server would be configured.

In the Act portion of the test, a request is made to the `TestServer` instance for the "/" path, and the response is read back into a string. This string is then compared with the expected string of "Hello World!". If they match, the test passes, otherwise it fails.

Now we can add a few additional integration tests to confirm that the prime checking functionality works via the web application:

```
public class PrimeWebCheckPrimeShould
{
    private readonly TestServer _server;
    private readonly HttpClient _client;
    public PrimeWebCheckPrimeShould()
    {
        // Arrange
        _server = new TestServer(new WebHostBuilder()
            .UseStartup<Startup>());
        _client = _server.CreateClient();
    }

    private async Task<string> GetCheckPrimeResponseString(
        string querystring = "")
    {
        var request = "/checkprime";
        if(!string.IsNullOrEmpty(querystring))
        {
            request += "?" + querystring;
        }
        var response = await _client.GetAsync(request);
        response.EnsureSuccessStatusCode();

        return await response.Content.ReadAsStringAsync();
    }

    [Fact]
    public async Task ReturnInstructionsGivenEmptyQueryString()
    {
        // Act
        var responseString = await GetCheckPrimeResponseString();
```

```
        // Assert
        Assert.Equal("Pass in a number to check in the form /checkprime?5",
            responseString);
    }
    [Fact]
    public async Task ReturnPrimeGiven5()
    {
        // Act
        var responseString = await GetCheckPrimeResponseString("5");

        // Assert
        Assert.Equal("5 is prime!",
            responseString);
    }

    [Fact]
    public async Task ReturnNotPrimeGiven6()
    {
        // Act
        var responseString = await GetCheckPrimeResponseString("6");

        // Assert
        Assert.Equal("6 is NOT prime!",
            responseString);
    }
}
```

Note that we're not really trying to test the correctness of our prime number checker with these tests, but rather that the web application is doing what we expect. We already have unit test coverage that gives us confidence in PrimeService, as you can see here:

Test Explorer

Search

Streaming Video: Improving quality with unit tests and fakes

Run All | Run... ▾ | Playlist : All Tests ▾

▲ **PrimeWeb.IntegrationTests** (4)
- ✓ PrimeWebCheckPrimeShould.ReturnInstructionsGivenEmptyQueryString
- ✓ PrimeWebCheckPrimeShould.ReturnNotPrimeGiven6
- ✓ PrimeWebCheckPrimeShould.ReturnPrimeGiven5
- ✓ PrimeWebDefaultRequestShould.ReturnHelloWorld

▲ **PrimeWeb.UnitTests** (11)
- ✓ PrimeService_IsPrimeShould.ReturnFalseGivenNonPrimesLessThan10(value: 4)
- ✓ PrimeService_IsPrimeShould.ReturnFalseGivenNonPrimesLessThan10(value: 6)
- ✓ PrimeService_IsPrimeShould.ReturnFalseGivenNonPrimesLessThan10(value: 8)
- ✓ PrimeService_IsPrimeShould.ReturnFalseGivenNonPrimesLessThan10(value: 9)
- ✓ PrimeService_IsPrimeShould.ReturnFalseGivenValuesLessThan2(value: 0)
- ✓ PrimeService_IsPrimeShould.ReturnFalseGivenValuesLessThan2(value: 1)
- ✓ PrimeService_IsPrimeShould.ReturnFalseGivenValuesLessThan2(value: -1)
- ✓ PrimeService_IsPrimeShould.ReturnTrueGivenPrimesLessThan10(value: 2)
- ✓ PrimeService_IsPrimeShould.ReturnTrueGivenPrimesLessThan10(value: 3)
- ✓ PrimeService_IsPrimeShould.ReturnTrueGivenPrimesLessThan10(value: 5)
- ✓ PrimeService_IsPrimeShould.ReturnTrueGivenPrimesLessThan10(value: 7)

: You can learn more about the unit tests in the Unit testing article.

Now that we have a set of passing tests, it's a good time to think about whether we're happy with the current way in which we've designed our application. If we see any code smells, now may be a good time to refactor the application to improve its design.

Refactoring to use Middleware

Refactoring is the process of changing an application's code to improve its design without changing its behavior. It should ideally be done when there is a suite of passing tests, since these help ensure the system's behavior remains the same before and after the changes. Looking at the way in which the prime checking logic is implemented in our web application, we see:

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.Run(async (context) =>
    {
```

```
        if (context.Request.Path.Value.Contains("checkprime"))
        {
            int numberToCheck;
            try
            {
                numberToCheck = int.Parse(context.Request.QueryString.Value.Replace("?",""));
                var primeService = new PrimeService();
                if (primeService.IsPrime(numberToCheck))
                {
                    await context.Response.WriteAsync(numberToCheck + " is prime!");
                }
                else
                {
                    await context.Response.WriteAsync(numberToCheck + " is NOT prime!");
                }
            }
            catch
            {
                await context.Response.WriteAsync("Pass in a number to check in the form /checkprim
            }
        }
        else
        {
            await context.Response.WriteAsync("Hello World!");
        }
    });
}
```

This code works, but it's far from how we would like to implement this kind of functionality in an ASP.NET Core application, even as simple a one as this is. Imagine what the `Configure` method would look like if we needed to add this much code to it every time we added another URL endpoint!

One option we can consider is adding *MVC* to the application, and creating a controller to handle the prime checking. However, assuming we don't currently need any other MVC functionality, that's a bit overkill.

We can, however, take advantage of ASP.NET Core *middleware*, which will help us encapsulate the prime checking logic in its own class and achieve better separation of concerns within the `Configure` method.

We want to allow the path the middleware uses to be specified as a parameter, so the middleware class expects a `RequestDelegate` and a `PrimeCheckerOptions` instance in its constructor. If the path of the request doesn't match what this middleware is configured to expect, we simply call the next middleware in the chain and do nothing further. The rest of the implementation code that was in `Configure` is now in the `Invoke` method.

: Since our middleware depends on the `PrimeService` service, we are also requesting an instance of this service via the constructor. The framework will provide this service via *Dependency Injection*, assuming it has been configured (e.g. in `ConfigureServices`).

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using PrimeWeb.Services;
using System;
using System.Threading.Tasks;

namespace PrimeWeb.Middleware
{
    public class PrimeCheckerMiddleware
```

```
    {
        private readonly RequestDelegate _next;
        private readonly PrimeCheckerOptions _options;
        private readonly PrimeService _primeService;

        public PrimeCheckerMiddleware(RequestDelegate next,
            PrimeCheckerOptions options,
            PrimeService primeService)
        {
            if (next == null)
            {
                throw new ArgumentNullException(nameof(next));
            }
            if (options == null)
            {
                throw new ArgumentNullException(nameof(options));
            }
            if (primeService == null)
            {
                throw new ArgumentNullException(nameof(primeService));
            }

            _next = next;
            _options = options;
            _primeService = primeService;
        }

        public async Task Invoke(HttpContext context)
        {
            var request = context.Request;
            if (!request.Path.HasValue ||
                request.Path != _options.Path)
            {
                await _next.Invoke(context);
            }
            else
            {
                int numberToCheck;
                if (int.TryParse(request.QueryString.Value.Replace("?", ""), out numberToCheck))
                {
                    if (_primeService.IsPrime(numberToCheck))
                    {
                        await context.Response.WriteAsync($"{numberToCheck} is prime!");
                    }
                    else
                    {
                        await context.Response.WriteAsync($"{numberToCheck} is NOT prime!");
                    }
                }
                else
                {
                    await context.Response.WriteAsync($"Pass in a number to check in the form {_optio
                }
            }
        }
    }
}
```

: Since this middleware acts as an endpoint in the request delegate chain when its path matches, there is no call to `_next.Invoke` in the case where this middleware handles the request.

With this middleware in place and some helpful extension methods created to make configuring it easier, the refactored `Configure` method looks like this:

```csharp
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UsePrimeChecker();

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

Following this refactoring, we are confident that the web application still works as before, since our integration tests are all passing.

: It's a good idea to commit your changes to source control after you complete a refactoring and your tests all pass. If you're practicing Test Driven Development, consider adding Commit to your Red-Green-Refactor cycle.

Summary

Integration testing provides a higher level of verification than unit testing. It tests application infrastructure and how different parts of an application work together. ASP.NET Core is very testable, and ships with a `TestServer` that makes wiring up integration tests for web server endpoints very easy.

Additional Resources

- Unit testing
- *Middleware*

1.7 Working with Data

1.7.1 Getting Started with ASP.NET Core and Entity Framework 6

By Paweł Grudzień and Damien Pontifex

This article will show you how to use Entity Framework 6 inside an ASP.NET Core application.

> **Sections:**
>
> - *Prerequisites*
> - *Setup connection strings and dependency injection*
> - *Migrate configuration from config to code*
> - *Summary*
> - *Additional Resources*

Prerequisites

Before you start, make sure that you compile against full .NET Framework in your project.json as Entity Framework 6 does not support .NET Core. If you need cross platform features you will need to upgrade to Entity Framework Core.

In your project.json file specify a single target for the full .NET Framework:

```
"frameworks": {
    "net46": {}
}
```

Setup connection strings and dependency injection

The simplest change is to explicitly get your connection string and setup dependency injection of your `DbContext` instance.

In your `DbContext` subclass, ensure you have a constructor which takes the connection string as so:

```
1  public class ApplicationDbContext : DbContext
2  {
3      public ApplicationDbContext(string nameOrConnectionString) : base(nameOrConnectionString)
4      {
5      }
6  }
```

In the `Startup` class within `ConfigureServices` add factory method of your context with it's connection string. Context should be resolved once per scope to ensure performance and ensure reliable operation of Entity Framework.

```
1  public void ConfigureServices(IServiceCollection services)
2  {
3      services.AddScoped(() => new ApplicationDbContext(Configuration["Data:DefaultConnection:Connectio
4
5      // Configure remaining services
6  }
```

Migrate configuration from config to code

Entity Framework 6 allows configuration to be specified in xml (in web.config or app.config) or through code. As of ASP.NET Core, all configuration is code-based.

Code-based configuration is achieved by creating a subclass of `System.Data.Entity.Config.DbConfiguration` and applying `System.Data.Entity.DbConfigurationTypeAttribute` to your `DbContext` subclass.

Our config file typically looked like this:

```
1  <entityFramework>
2      <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory, Entit
3          <parameters>
4              <parameter value="mssqllocaldb" />
5          </parameters>
6      </defaultConnectionFactory>
7      <providers>
8          <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProvide
9      </providers>
10 </entityFramework>
```

The `defaultConnectionFactory` element sets the factory for connections. If this attribute is not set then the default value is `SqlConnectionProvider`. If, on the other hand, value is provided, the given class will be used to create `DbConnection` with its `CreateConnection` method. If the given factory has no default constructor then you must add parameters that are used to construct the object.

```
1  [DbConfigurationType(typeof(CodeConfig))] // point to the class that inherit from DbConfiguration
2  public class ApplicationDbContext : DbContext
3  {
4      [...]
5  }
6
7  public class CodeConfig : DbConfiguration
8  {
9      public CodeConfig()
10     {
11         SetProviderServices("System.Data.SqlClient",
12             System.Data.Entity.SqlServer.SqlProviderServices.Instance);
13     }
14 }
```

SQL Server, SQL Server Express and LocalDB

This is the default and so no explicit configuration is needed. The above `CodeConfig` class can be used to explicitly set the provider services and the appropriate connection string should be passed to the `DbContext` constructor as shown *above*.

Summary

Entity Framework 6 is an object relational mapping (ORM) library, that is capable of mapping your classes to database entities with little effort. These features made it very popular so migrating large portions of code may be undesirable for many projects. This article shows how to avoid migration to focus on other new features of ASP.NET.

Additional Resources

- Entity Framework - Code-Based Configuration

1.7.2 Azure Storage

1.8 Client-Side Development

1.8.1 Using Gulp

By Erik Reitan, Scott Addie and Daniel Roth

In a typical modern web application, the build process might:

- Bundle and minify JavaScript and CSS files.

- Run tools to call the bundling and minification tasks before each build.

- Compile LESS or SASS files to CSS.

- Compile CoffeeScript or TypeScript files to JavaScript.

A *task runner* is a tool which automates these routine development tasks and more. Visual Studio provides built-in support for two popular JavaScript-based task runners: Gulp and Grunt.

Sections:

- *Introducing Gulp*
- *Gulp Starter Tasks in Visual Studio*
- *Running Default Tasks*
- *Defining and Running a New Task*
- *Defining and Running Tasks in a Series*
- *IntelliSense*
- *Development, Staging, and Production Environments*
- *Switching Between Environments*
- *Task and Module Details*
- *Gulp Recipes*
- *Summary*
- *See Also*

Introducing Gulp

Gulp is a JavaScript-based streaming build toolkit for client-side code. It is commonly used to stream client-side files through a series of processes when a specific event is triggered in a build environment. Some advantages of using Gulp include the automation of common development tasks, the simplification of repetitive tasks, and a decrease in overall development time. For instance, Gulp can be used to automate *bundling and minification* or the cleansing of a development environment before a new build.

The ASP.NET Core Web Application project template is used to help you get started designing and coding a new Web application in Visual Studio. It contains default functionality to demonstrate many aspects of ASP.NET. The template also includes Node Package Manager (npm) and Gulp, making it easier to add bundling and minification to a project.

: You don't need the ASP.NET Core Web Application project template or Visual Studio to implement bundling and minification. For example, create an ASP.NET project using Yeoman, push it to GitHub, clone it on a Mac, and then bundle and minify the project.

When you create a new web project using ASP.NET Core Web Application template, Visual Studio includes the Gulp.js npm package, the *gulpfile.js* file, and a set of Gulp dependencies. The npm package contains all the prerequisites for

---

running Gulp tasks in your Visual Studio project. The provided *gulpfile.js* file defines a set of Gulp tasks which can be run from the **Task Runner Explorer** window in Visual Studio. The devDependencies section of the *package.json* file specifies the development-time dependencies to install. These dependencies are not deployed with the application. You can add new packages to devDependencies and save the file:

```
{
  "name": "ASP.NET",
  "version": "0.0.0",
  "devDependencies": {
  "gulp": "3.8.11",
  "gulp-concat": "2.5.2",
  "gulp-cssmin": "0.1.7",
  "gulp-uglify": "1.2.0",
  "rimraf": "2.2.8"
  }
}
```

After adding a new key-value pair in devDependencies and saving the file, Visual Studio will download and install the corresponding version of the package. In **Solution Explorer**, these packages are found in **Dependencies > npm**.

Gulp Starter Tasks in Visual Studio

A starter set of Gulp tasks is defined in *gulpfile.js*. These tasks delete and minify the CSS and JavaScript files. The following JavaScript, from the first half of *gulpfile.js*, includes Gulp modules and specifies file paths to be referenced within the forthcoming tasks:

```
/// <binding Clean='clean' />
"use strict";

var gulp = require("gulp"),
  rimraf = require("rimraf"),
  concat = require("gulp-concat"),
  cssmin = require("gulp-cssmin"),
  uglify = require("gulp-uglify");

var paths = {
  webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.minJs = paths.webroot + "js/**/*.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.minCss = paths.webroot + "css/**/*.min.css";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.concatCssDest = paths.webroot + "css/site.min.css";
```

The above code specifies which Node modules are required. The require function imports each module so that the dependent tasks can utilize their features. Each of the imported modules is assigned to a variable. The modules can be located either by name or path. In this example, the modules named gulp, rimraf, gulp-concat, gulp-cssmin, and gulp-uglify are retrieved by name. Additionally, a series of paths are created so that the locations of CSS and JavaScript files can be reused and referenced within the tasks. The following table provides descriptions of the modules included in *gulpfile.js*.

Module Name	Description
gulp	The Gulp streaming build system. For more information, see gulp.
rimraf	A Node deletion module. For more information, see rimraf.
gulp-concat	A module that will concatenate files based on the operating system's newline character. For more information, see gulp-concat.
gulp-cssmin	A module that will minify CSS files. For more information, see gulp-cssmin.
gulp-uglify	A module that minifies *.js* files using the UglifyJS toolkit. For more information, see gulp-uglify.

Once the requisite modules are imported in *gulpfile.js*, the tasks can be specified. Visual Studio registers six tasks, represented by the following code in *gulpfile.js*:

```
gulp.task("clean:js", function (cb) {
  rimraf(paths.concatJsDest, cb);
});

gulp.task("clean:css", function (cb) {
  rimraf(paths.concatCssDest, cb);
});

gulp.task("clean", ["clean:js", "clean:css"]);

gulp.task("min:js", function () {
  return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
    .pipe(concat(paths.concatJsDest))
    .pipe(uglify())
    .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
  return gulp.src([paths.css, "!" + paths.minCss])
    .pipe(concat(paths.concatCssDest))
    .pipe(cssmin())
    .pipe(gulp.dest("."));
});

gulp.task("min", ["min:js", "min:css"]);
```

The following table provides an explanation of the tasks specified in the code above:

Task Name	Description
clean:js	A task that uses the rimraf Node deletion module to remove the minified version of the *site.js* file.
clean:css	A task that uses the rimraf Node deletion module to remove the minified version of the *site.css* file.
clean	A task that calls the `clean:js` task, followed by the `clean:css` task.
min:js	A task that minifies and concatenates all *.js* files within the *js* folder. The *.min.js* files are excluded.
min:css	A task that minifies and concatenates all *.css* files within the *css* folder. The *.min.css* files are excluded.
min	A task that calls the `min:js` task, followed by the `min:css` task.

Running Default Tasks

If you haven't already created a new Web app, create a new ASP.NET Web Application project in Visual Studio.

1. Select **File > New > Project** from the menu bar. The **New Project** dialog box is displayed.
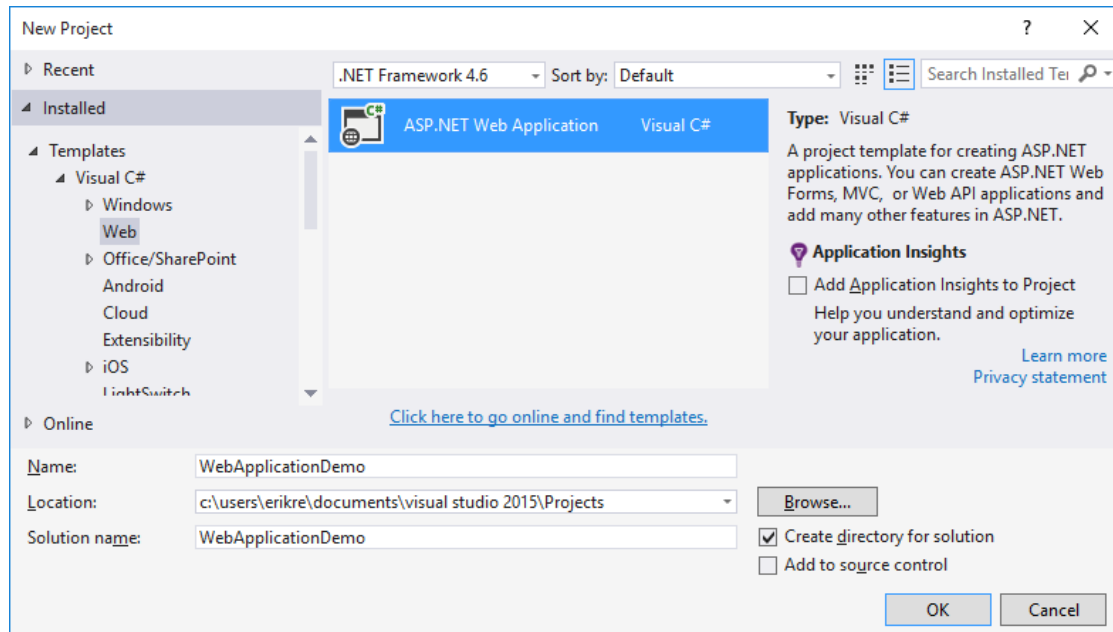
2. Select the **ASP.NET Web Application** template, choose a project name, and click **OK**.

3. In the **New ASP.NET Project** dialog box, select the ASP.NET Core **Web Application** template and click **OK**.

4. In **Solution Explorer**, right-click *gulpfile.js*, and select **Task Runner Explorer**.
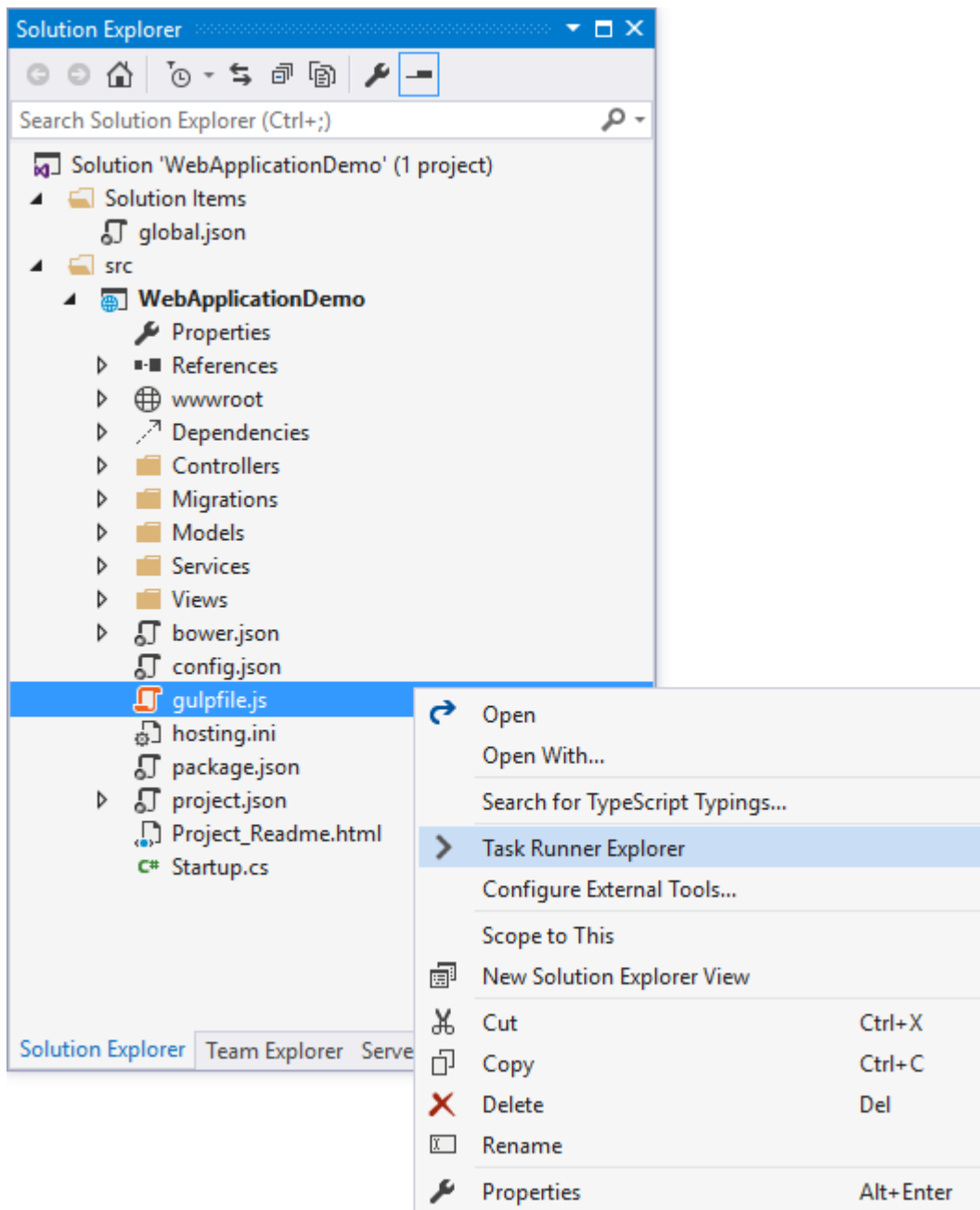
Task Runner Explorer shows the list of Gulp tasks. In the default ASP.NET Core Web Application template in Visual Studio, there are six tasks included from *gulpfile.js*.
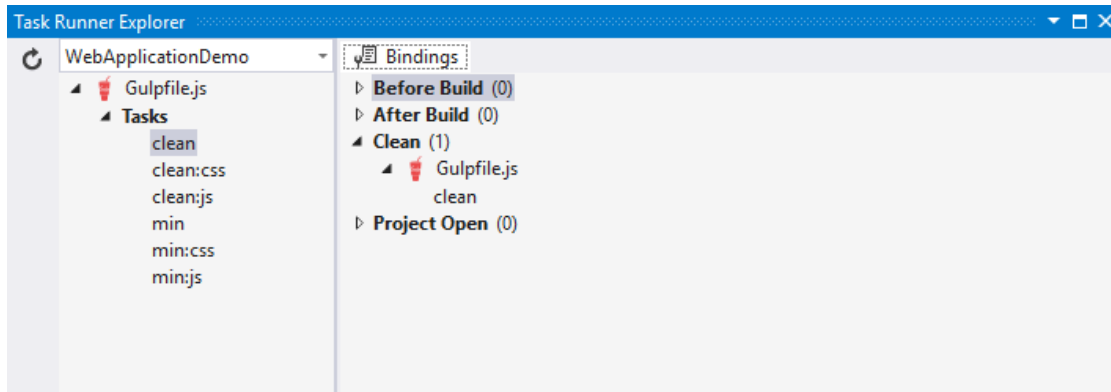
5. Underneath **Tasks** in **Task Runner Explorer**, right-click **clean**, and select **Run** from the pop-up menu.
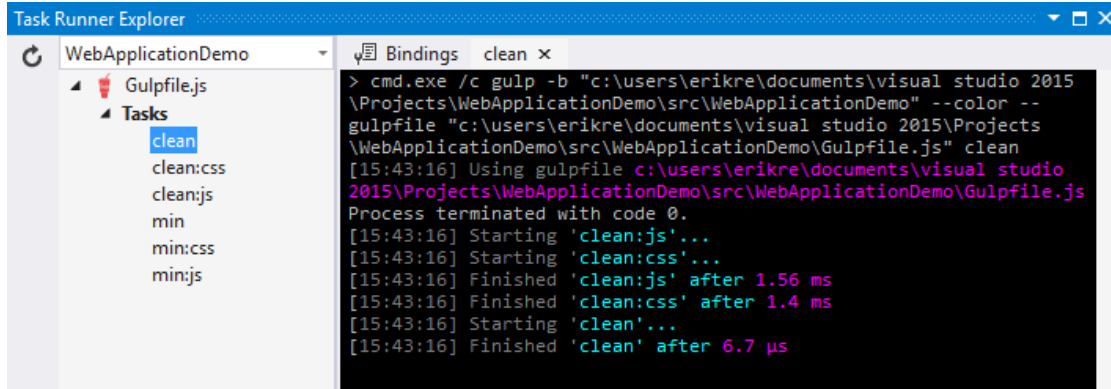
Task Runner Explorer will create a new tab named **clean** and execute the related clean task as it is defined in *gulpfile.js*.

6. Next, right-click the **clean** task, then select **Bindings > Before Build**.
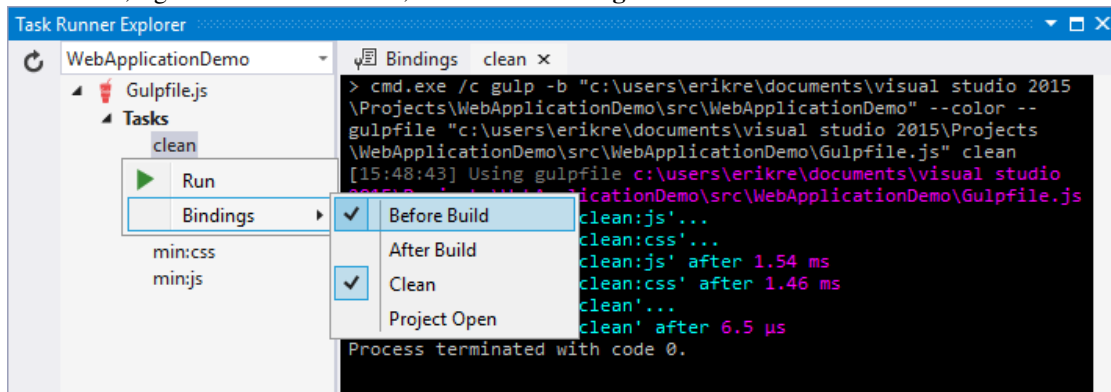
The **Before Build** binding option allows the clean task to run automatically before each build of the project.

It's worth noting that the bindings you set up with **Task Runner Explorer** are **not** stored in the *project.json*. Rather they are stored in the form of a comment at the top of your *gulpfile.js*. It is possible (as demonstrated in the default project templates) to have gulp tasks kicked off by the *scripts* section of your *project.json*. **Task Runner Explorer** is a way you can configure tasks to run using Visual Studio. If you are using a different editor (for example, Visual Studio Code) then using the *project.json* will probably be the most straightforward way to bring together the various stages (prebuild, build, etc.) and the running of gulp tasks.

: *project.json* stages are not triggered when building in Visual Studio by default. If you want to ensure that they are set this option in the Visual Studio project properties: Build tab -> Produce outputs on build. This will add a *ProduceOutputsOnBuild* element to your *.xproj* file which will cause Visual studio to trigger the *project.json* stages when building.

Defining and Running a New Task

To define a new Gulp task, modify *gulpfile.js*.

1. Add the following JavaScript to the end of *gulpfile.js*:

```javascript
gulp.task("first", function () {
  console.log('first task! <----');
});
```

This task is named `first`, and it simply displays a string.

2. Save *gulpfile.js*.

3. In **Solution Explorer**, right-click *gulpfile.js*, and select *Task Runner Explorer*.

4. In **Task Runner Explorer**, right-click **first**, and select **Run**.
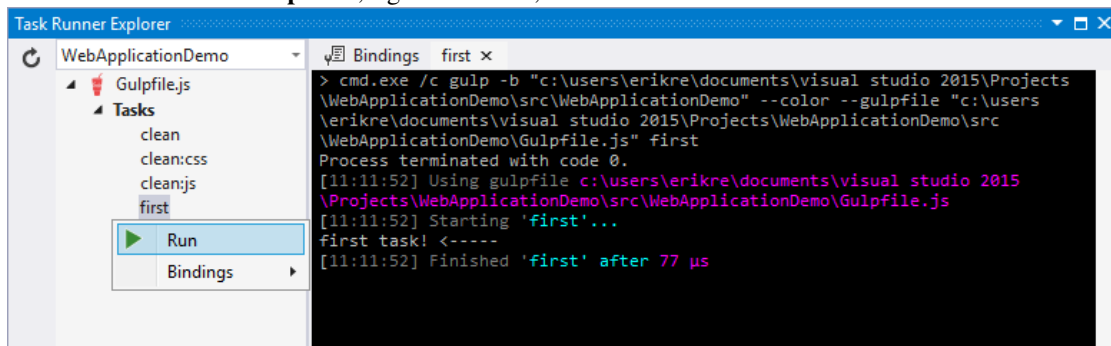
You'll see that the output text is displayed. If you are interested in examples based on a common scenario, see Gulp Recipes.

Defining and Running Tasks in a Series

When you run multiple tasks, the tasks run concurrently by default. However, if you need to run tasks in a specific order, you must specify when each task is complete, as well as which tasks depend on the completion of another task.

1. To define a series of tasks to run in order, replace the `first` task that you added above in *gulpfile.js* with the following:

```javascript
gulp.task("series:first", function () {
  console.log('first task! <----');
});

gulp.task("series:second", ["series:first"], function () {
  console.log('second task! <----');
});

gulp.task("series", ["series:first", "series:second"], function () {});
```

You now have three tasks: `series:first`, `series:second`, and `series`. The `series:second` task includes a second parameter which specifies an array of tasks to be run and completed before the `series:second` task will run. As specified in the code above, only the `series:first` task must be completed before the `series:second` task will run.

2. Save *gulpfile.js*.

3. In **Solution Explorer**, right-click *gulpfile.js* and select **Task Runner Explorer** if it isn't already open.

4. In **Task Runner Explorer**, right-click **series** and select **Run**.
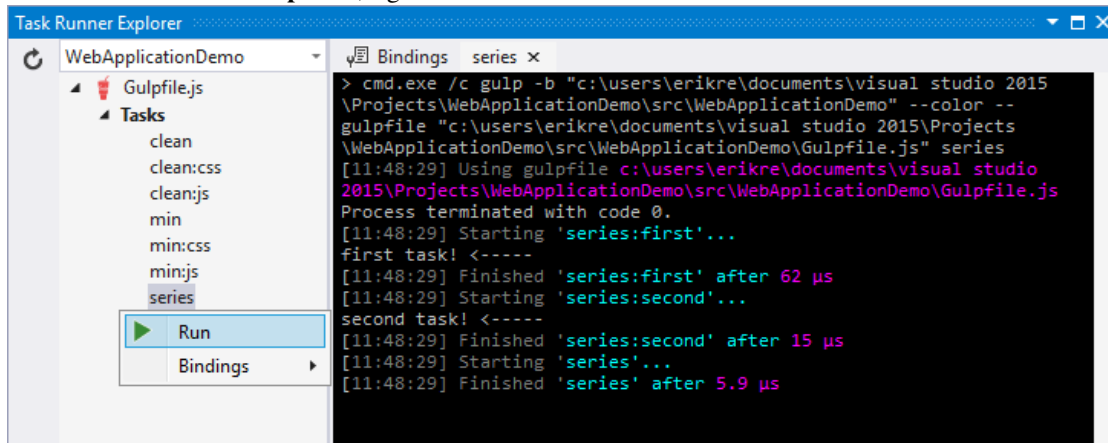
IntelliSense

IntelliSense provides code completion, parameter descriptions, and other features to boost productivity and to decrease errors. Gulp tasks are written in JavaScript; therefore, IntelliSense can provide assistance while developing. As you work with JavaScript, IntelliSense lists the objects, functions, properties, and parameters that are available based on your current context. Select a coding option from the pop-up list provided by IntelliSense to complete the code.
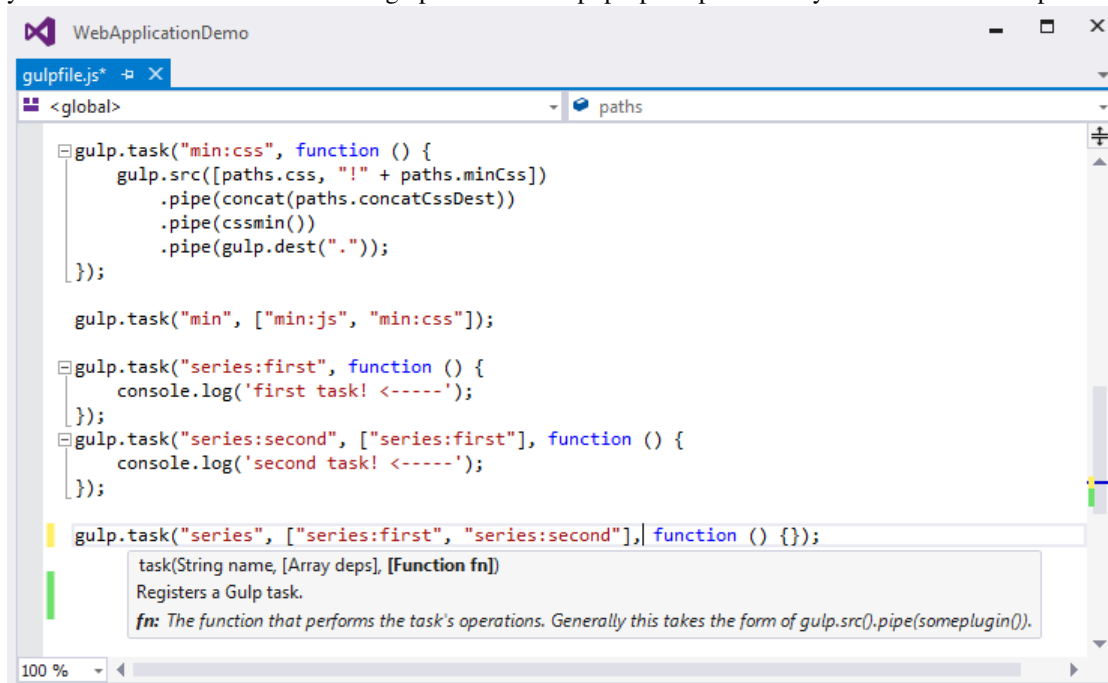
For more information about IntelliSense, see JavaScript IntelliSense.

Development, Staging, and Production Environments

When Gulp is used to optimize client-side files for staging and production, the processed files are saved to a local staging and production location. The *_Layout.cshtml* file uses the **environment** tag helper to provide two different versions of CSS files. One version of CSS files is for development and the other version is optimized for both staging and production. In Visual Studio 2015, when you change the **Hosting:Environment** environment variable to `Production`, Visual Studio will build the Web app and link to the minimized CSS files. The following markup shows the **environment** tag helpers containing link tags to the `Development` CSS files and the minified `Staging`, `Production` CSS files.

```
<environment names="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment names="Staging,Production">
  <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/css/bootstrap.min.css"
      asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
      asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-value
  <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

Switching Between Environments

To switch between compiling for different environments, modify the **Hosting:Environment** environment variable's value.

1. In **Task Runner Explorer**, verify that the **min** task has been set to run **Before Build**.

2. In **Solution Explorer**, right-click the project name and select **Properties**.

   The property sheet for the Web app is displayed.

3. Click the **Debug** tab.

4. Set the value of the **Hosting:Environment** environment variable to `Production`.

5. Press **F5** to run the application in a browser.

6. In the browser window, right-click the page and select **View Source** to view the HTML for the page.

   Notice that the stylesheet links point to the minified CSS files.

7. Close the browser to stop the Web app.

8. In Visual Studio, return to the property sheet for the Web app and change the **Hosting:Environment** environment variable back to `Development`.

9. Press **F5** to run the application in a browser again.

10. In the browser window, right-click the page and select **View Source** to see the HTML for the page.

    Notice that the stylesheet links point to the unminified versions of the CSS files.

For more information related to environments in ASP.NET Core, see *Working with Multiple Environments*.

Task and Module Details

A Gulp task is registered with a function name. You can specify dependencies if other tasks must run before the current task. Additional functions allow you to run and watch the Gulp tasks, as well as set the source (*src*) and destination (*dest*) of the files being modified. The following are the primary Gulp API functions:

Gulp Func- tion	Syntax	Description
task	`gulp.task(name[, deps], fn) { }`	The `task` function creates a task. The `name` parameter defines the name of the task. The `deps` parameter contains an array of tasks to be completed before this task runs. The `fn` parameter represents a callback function which performs the operations of the task.
watch	`gulp.watch(glob [, opts], tasks) { }`	The `watch` function monitors files and runs tasks when a file change occurs. The `glob` parameter is a `string` or `array` that determines which files to watch. The `opts` parameter provides additional file watching options.
src	`gulp.src(globs[, options]) { }`	The `src` function provides files that match the `glob` value(s). The `glob` parameter is a `string` or `array` that determines which files to read. The `options` parameter provides additional file options.
dest	`gulp.dest(path[, options]) { }`	The `dest` function defines a location to which files can be written. The `path` parameter is a string or function that determines the destination folder. The `options` parameter is an object that specifies output folder options.

For additional Gulp API reference information, see Gulp Docs API.

Gulp Recipes

The Gulp community provides Gulp recipes. These recipes consist of Gulp tasks to address common scenarios.

Summary

Gulp is a JavaScript-based streaming build toolkit that can be used for bundling and minification. Visual Studio automatically installs Gulp along with a set of Gulp plugins. Gulp is maintained on GitHub. For additional information about Gulp, see the Gulp Documentation on GitHub.

See Also

- *Bundling and Minification*
- *Using Grunt*

1.8.2 Using Grunt

Noel Rice

Grunt is a JavaScript task runner that automates script minification, TypeScript compilation, code quality "lint" tools, CSS pre-processors, and just about any repetitive chore that needs doing to support client development. Grunt is fully supported in Visual Studio, though the ASP.NET project templates use Gulp by default (see *Using Gulp*).

Sections:

This example uses an empty ASP.NET Core project as its starting point, to show how to automate the client build process from scratch.

The finished example cleans the target deployment directory, combines JavaScript files, checks code quality, condenses JavaScript file content and deploys to the root of your web application. We will use the following packages:

- **grunt**: The Grunt task runner package.

- **grunt-contrib-clean**: A plugin that removes files or directories.

- **grunt-contrib-jshint**: A plugin that reviews JavaScript code quality.

- **grunt-contrib-concat**: A plugin that joins files into a single file.

- **grunt-contrib-uglify**: A plugin that minifies JavaScript to reduce size.

- **grunt-contrib-watch**: A plugin that watches file activity.

Preparing the application

To begin, set up a new empty web application and add TypeScript example files. TypeScript files are automatically compiled into JavaScript using default Visual Studio settings and will be our raw material to process using Grunt.

1. In Visual Studio, create a new `ASP.NET Web Application`.

2. In the **New ASP.NET Project** dialog, select the ASP.NET Core **Empty** template and click the OK button.

3. In the Solution Explorer, review the project structure. The `\src` folder includes empty `wwwroot` and `Dependencies` nodes.
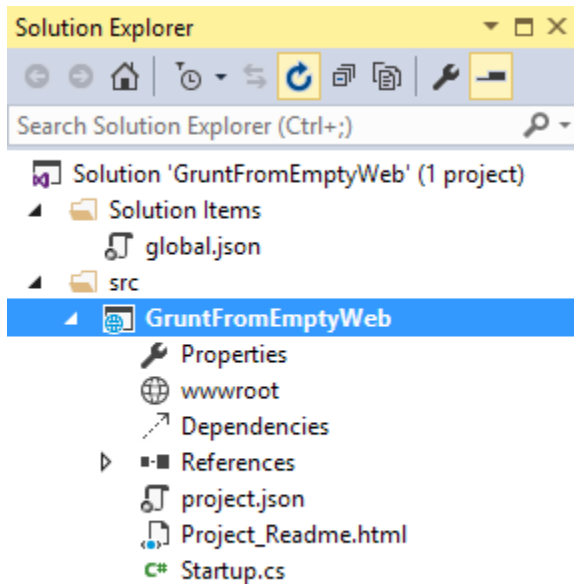
4. Add a new folder named `TypeScript` to your project directory.

5. Before adding any files, let's make sure that Visual Studio has the option 'compile on save' for TypeScript files checked. *Tools > Options > Text Editor > Typescript > Project*
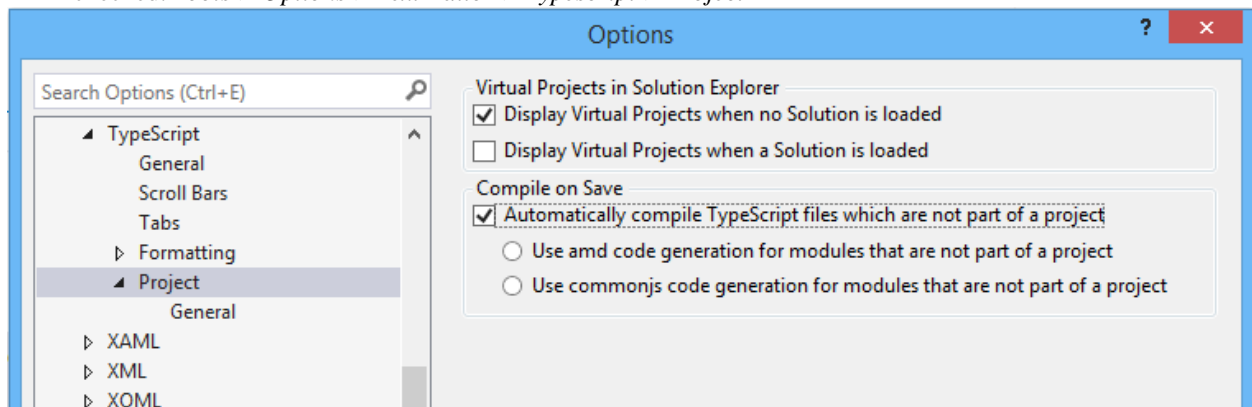
6. Right-click the `TypeScript` directory and select **Add > New Item** from the context menu. Select the **JavaScript file** item and name the file **Tastes.ts** (note the *.ts extension). Copy the line of TypeScript code below into the file (when you save, a new Tastes.js file will appear with the JavaScript source).

```
enum Tastes { Sweet, Sour, Salty, Bitter }
```

7. Add a second file to the **TypeScript** directory and name it `Food.ts`. Copy the code below into the file.

```
class Food {
  constructor(name: string, calories: number) {
    this._name = name;
    this._calories = calories;
  }

  private _name: string;
  get Name() {
    return this._name;
```

```
  }

  private _calories: number;
  get Calories() {
    return this._calories;
  }

  private _taste: Tastes;
  get Taste(): Tastes { return this._taste }
  set Taste(value: Tastes) {
    this._taste = value;
  }
}
```

Configuring NPM

Next, configure NPM to download grunt and grunt-tasks.

1. In the Solution Explorer, right-click the project and select **Add > New Item** from the context menu. Select the **NPM configuration file** item, leave the default name, `package.json`, and click the **Add** button.

2. In the package.json file, inside the `devDependencies` object braces, enter "grunt". Select `grunt` from the Intellisense list and press the Enter key. Visual Studio will quote the grunt package name, and add a colon. To the right of the colon, select the latest stable version of the package from the top of the Intellisense list (press `Ctrl-Space` if Intellisense does not appear).

```
"devDependencies": {
    "grunt": ""
}
                    ⊞  0.4.5
                    ⊞  ^0.4.5
                    ⊞  ~0.4.5
```
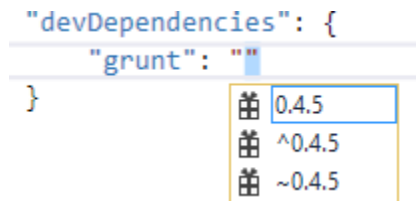
: NPM uses semantic versioning to organize dependencies. Semantic versioning, also known as SemVer, identifies packages with the numbering scheme <major>.<minor>.<patch>. Intellisense simplifies semantic versioning by showing only a few common choices. The top item in the Intellisense list (0.4.5 in the example above) is considered the latest stable version of the package. The caret (^) symbol matches the most recent major version and the tilde (~) matches the most recent minor version. See the NPM semver version parser reference as a guide to the full expressivity that SemVer provides.

---

3. Add more dependencies to load grunt-contrib* packages for *clean, jshint, concat, uglify and watch* as shown in the example below. The versions do not need to match the example.

```
"devDependencies": {
    "grunt": "0.4.5",
    "grunt-contrib-clean": "0.6.0",
    "grunt-contrib-jshint": "0.11.0",
    "grunt-contrib-concat": "0.5.1",
    "grunt-contrib-uglify": "0.8.0",
    "grunt-contrib-watch": "0.6.1"
}
```

4. Save the `package.json` file.

The packages for each devDependencies item will download, along with any files that each package requires. You can find the package files in the `node_modules` directory by enabling the **Show All Files** button in the Solution Explorer.

- ◢ 🗀 node_modules
  - ▷ 🗀 grunt
  - ▷ 🗀 grunt-contrib-clean
  - ▷ 🗀 grunt-contrib-concat
  - ▷ 🗀 grunt-contrib-jshint
  - ▷ 🗀 grunt-contrib-uglify
  - ▷ 🗀 grunt-contrib-watch
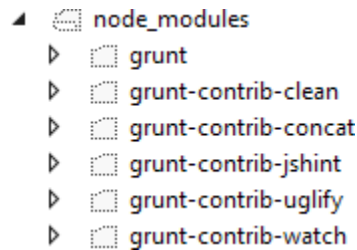
: If you need to, you can manually restore dependencies in Solution Explorer by right-clicking on `Dependencies\NPM` and selecting the **Restore Packages** menu option.
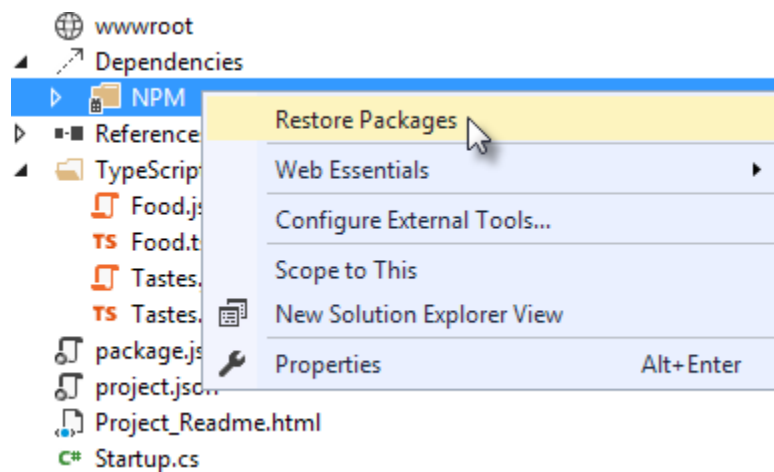
Configuring Grunt

Grunt is configured using a manifest named `Gruntfile.js` that defines, loads and registers tasks that can be run manually or configured to run automatically based on events in Visual Studio.

1. Right-click the project and select **Add > New Item**. Select the **Grunt Configuration file** option, leave the default name, `Gruntfile.js`, and click the **Add** button.

The initial code includes a module definition and the `grunt.initConfig()` method. The `initConfig()` is used to set options for each package, and the remainder of the module will load and register tasks.

```
module.exports = function (grunt) {
  grunt.initConfig({
  });
};
```

2. Inside the `initConfig()` method, add options for the `clean` task as shown in the example Gruntfile.js below. The clean task accepts an array of directory strings. This task removes files from wwwroot/lib and removes the entire /temp directory.

```
module.exports = function (grunt) {
  grunt.initConfig({
    clean: ["wwwroot/lib/*", "temp/"],
```

```
    });
};
```

3. Below the initConfig() method, add a call to `grunt.loadNpmTasks()`. This will make the task runnable from Visual Studio.

```
grunt.loadNpmTasks("grunt-contrib-clean");
```

4. Save Gruntfile.js. The file should look something like the screenshot below.
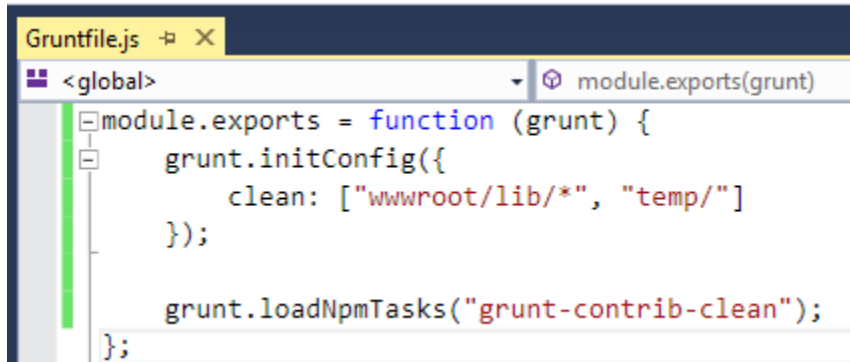
5. Right-click Gruntfile.js and select **Task Runner Explorer** from the context menu. The Task Runner Explorer window will open.
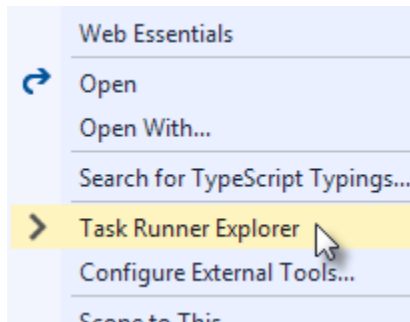
6. Verify that `clean` shows under **Tasks** in the Task Runner Explorer.
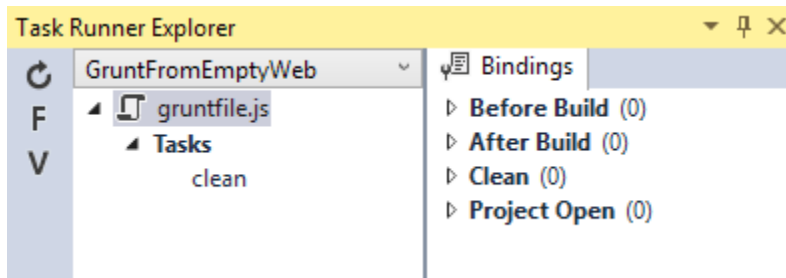
7. Right-click the clean task and select **Run** from the context menu. A command window displays progress of the task.
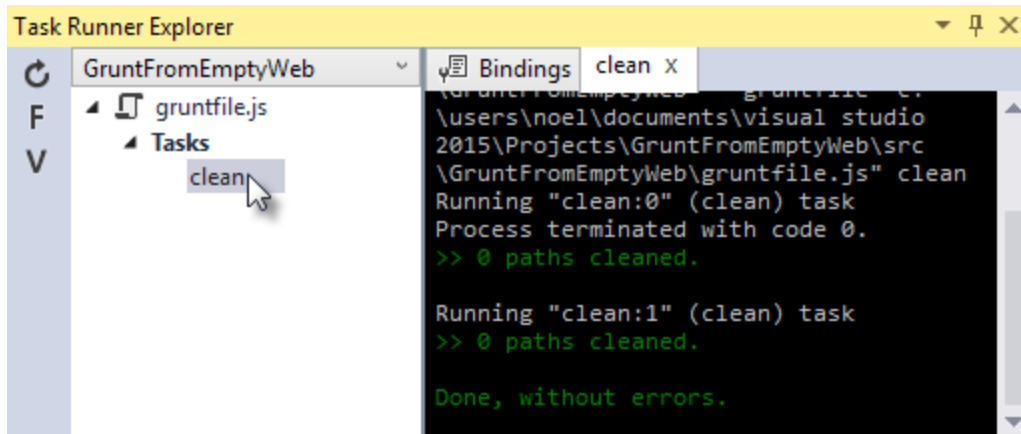
: There are no files or directories to clean yet. If you like, you can manually create them in the Solution Explorer and then run the clean task as a test.

8. In the initConfig() method, add an entry for `concat` using the code below.

The `src` property array lists files to combine, in the order that they should be combined. The `dest` property assigns the path to the combined file that is produced.

```
concat: {
  all: {
    src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
    dest: 'temp/combined.js'
  }
},
```

: The `all` property in the code above is the name of a target. Targets are used in some Grunt tasks to allow multiple build environments. You can view the built-in targets using Intellisense or assign your own.

9. Add the `jshint` task using the code below.

The jshint code-quality utility is run against every JavaScript file found in the temp directory.

```
jshint: {
  files: ['temp/*.js'],
  options: {
    '-W069': false,
  }
},
```

: The option "-W069" is an error produced by jshint when JavaScript uses bracket syntax to assign a property instead of dot notation, i.e. `Tastes["Sweet"]` instead of `Tastes.Sweet`. The option turns off the warning to allow the rest of the process to continue.

10. Add the `uglify` task using the code below.

The task minifies the combined.js file found in the temp directory and creates the result file in wwwroot/lib following the standard naming convention <file name>.min.js.

```
uglify: {
  all: {
    src: ['temp/combined.js'],
    dest: 'wwwroot/lib/combined.min.js'
  }
},
```

11. Under the call grunt.loadNpmTasks() that loads grunt-contrib-clean, include the same call for jshint, concat and uglify using the code below.

```
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');
```

12. Save `Gruntfile.js`. The file should look something like the example below.

13. Notice that the Task Runner Explorer Tasks list includes `clean`, `concat`, `jshint` and `uglify` tasks. Run each task in order and observe the results in Solution Explorer. Each task should run without errors.
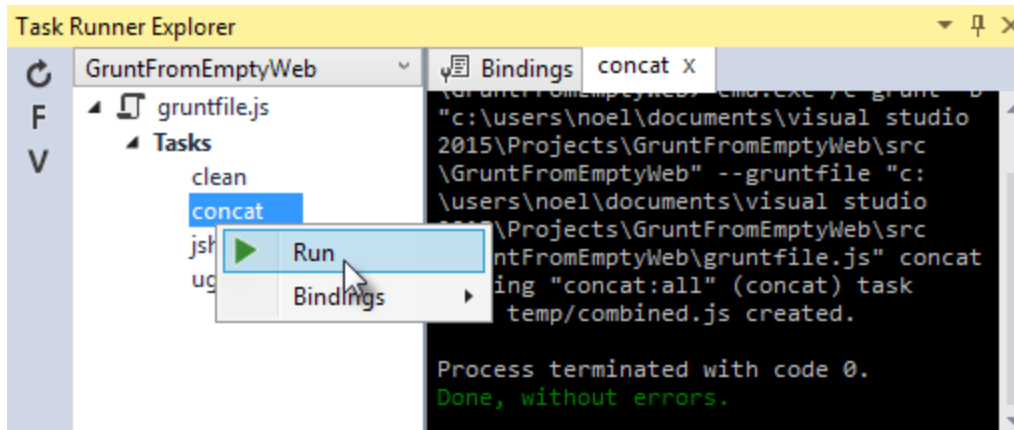
The concat task creates a new combined.js file and places it into the temp directory. The jshint task simply runs and doesn't produce output. The uglify task creates a new combined.min.js file and places it into wwwroot/lib. On completion, the solution should look something like the screenshot below:
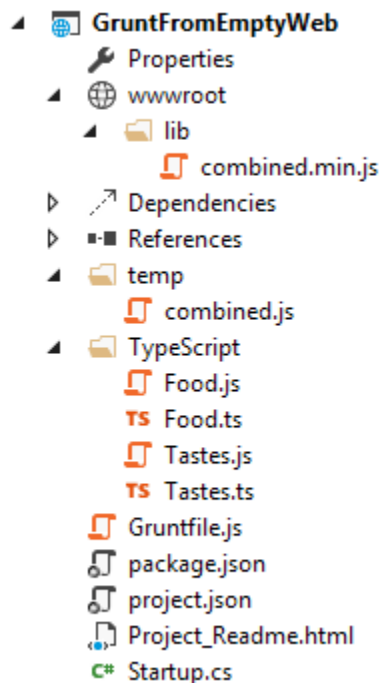
: For more information on the options for each package, visit https://www.npmjs.com/ and lookup the package name in the search box on the main page. For example, you can look up the grunt-contrib-clean package to get a documentation link that explains all of its parameters.

All Together Now

Use the Grunt `registerTask()` method to run a series of tasks in a particular sequence. For example, to run the example steps above in the order clean -> concat -> jshint -> uglify, add the code below to the module. The code should be added to the same level as the loadNpmTasks() calls, outside initConfig.

```
grunt.registerTask("all", ['clean', 'concat', 'jshint', 'uglify']);
```

The new task shows up in Task Runner Explorer under Alias Tasks. You can right-click and run it just as you would other tasks. The `all` task will run `clean`, `concat`, `jshint` and `uglify`, in order.
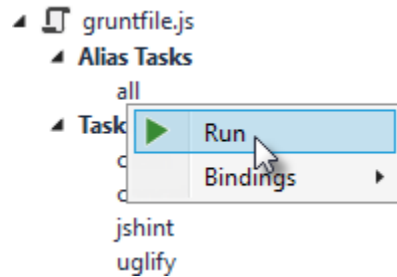
Watching for changes
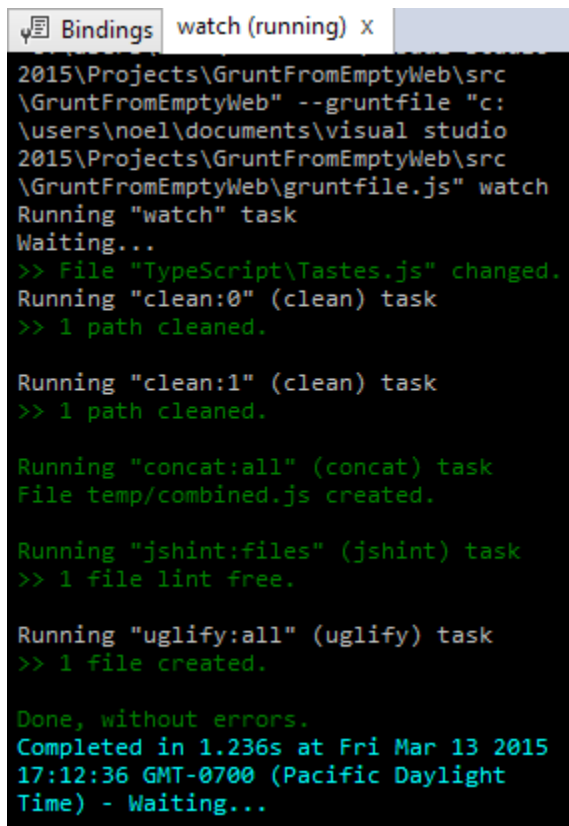
A `watch` task keeps an eye on files and directories. The watch triggers tasks automatically if it detects changes. Add the code below to initConfig to watch for changes to *.js files in the TypeScript directory. If a JavaScript file is changed, `watch` will run the `all` task.

```
watch: {
  files: ["TypeScript/*.js"],
  tasks: ["all"]
}
```

Add a call to `loadNpmTasks()` to show the `watch` task in Task Runner Explorer.

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

Right-click the watch task in Task Runner Explorer and select Run from the context menu. The command window that shows the watch task running will display a "Waiting..." message. Open one of the TypeScript files, add a space, and then save the file. This will trigger the watch task and trigger the other tasks to run in order. The screenshot below shows a sample run.

Binding to Visual Studio Events

Unless you want to manually start your tasks every time you work in Visual Studio, you can bind tasks to **Before Build**, **After Build**, **Clean**, and **Project Open** events.

Let's bind `watch` so that it runs every time Visual Studio opens. In Task Runner Explorer, right-click the watch task and select **Bindings > Project Open** from the context menu.
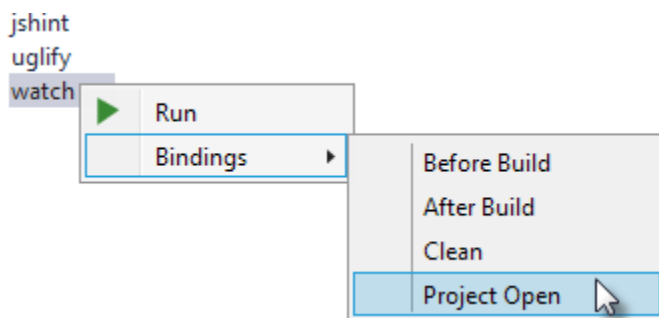
Unload and reload the project. When the project loads again, the watch task will start running automatically.

Summary

Grunt is a powerful task runner that can be used to automate most client-build tasks. Grunt leverages NPM to deliver its packages, and features tooling integration with Visual Studio. Visual Studio's Task Runner Explorer detects changes to configuration files and provides a convenient interface to run tasks, view running tasks, and bind tasks to Visual Studio events.

See Also

- *Using Gulp*

1.8.3 Manage Client-Side Packages with Bower

By Noel Rice, Scott Addie

Bower is a "package manager for the web." Bower lets you install and restore client-side packages, including JavaScript and CSS libraries. For example, with Bower you can install CSS files, fonts, client frameworks, and JavaScript libraries from external sources. Bower resolves dependencies and will automatically download and install all the packages you need. For example, if you configure Bower to load the Bootstrap package, the necessary jQuery package will automatically come along for the ride. For .NET libraries you still use NuGet package manager.

: Visual Studio developers are already familiar with NuGet, so why not use NuGet instead of Bower? Mainly because Bower already has a rich ecosystem with over 34,000 packages in play; and, it integrates well with the Gulp and Grunt task runners.

Getting Started with Bower

ASP.NET Core project templates pre-construct the client build process for you. The ubiquitous jQuery and Bootstrap packages are installed, and the plumbing for NPM, Gulp, and Bower is already in place. The screenshot below depicts the initial project in Solution Explorer. It's important to enable the "Show All Files" option, as the bower.json file is hidden by default.

Solution Explorer

Search Solution Explorer (Ctrl+

Solution 'Starter' (1 project)

▷ 📁 Solution Items

◢ 📁 src

◢ 🌐 **Starter**

   ▷ 🔧 Properties

   ▷ ∎-∎ References

   ▷ 🌐 wwwroot

   ▷ ↗ Dependencies

   ▷ 📁 Controllers

   ▷ ⬚ node_modules

   ▷ 📁 Views

   🎵 appsettings.json
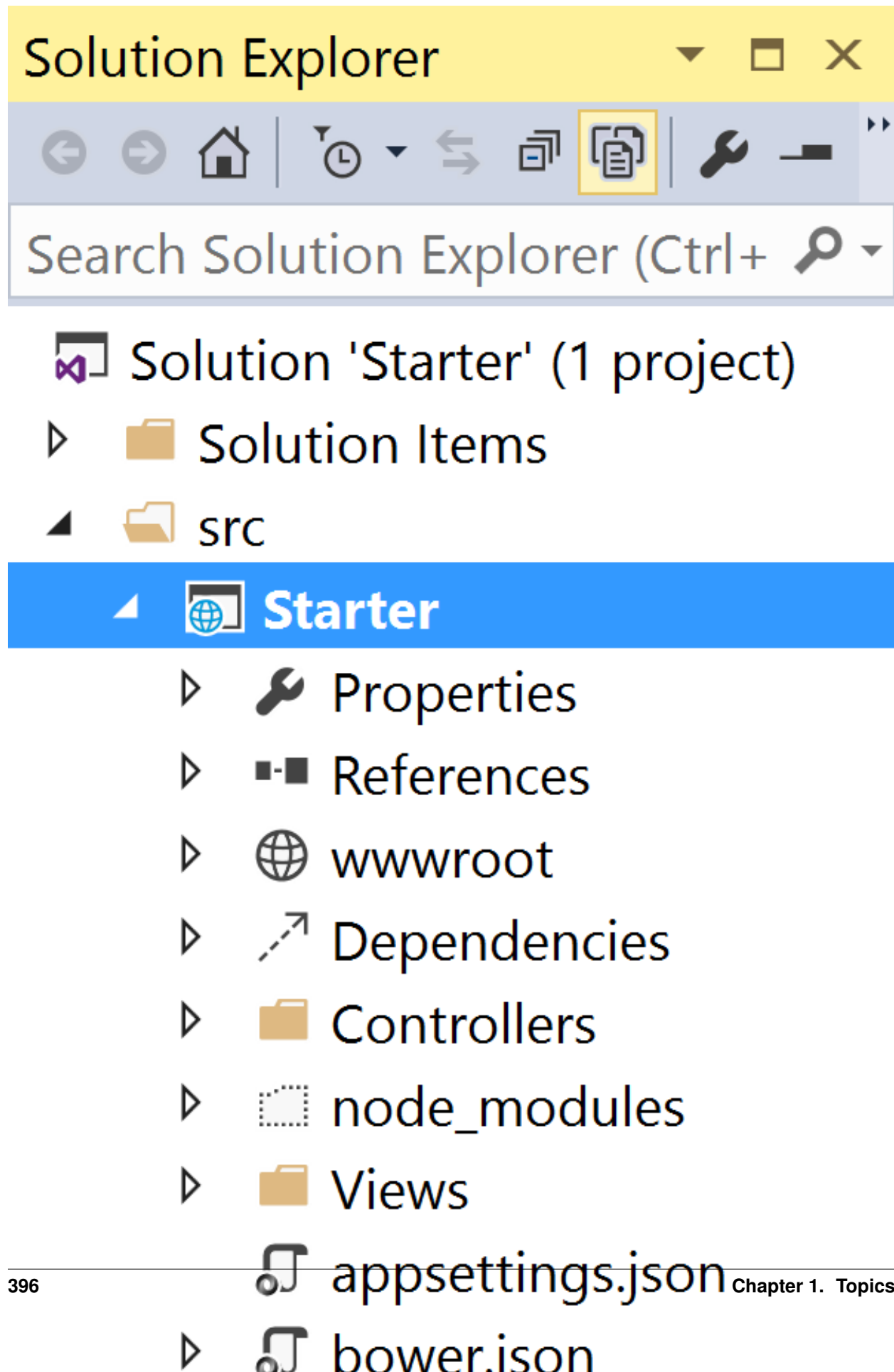
   ▷ 🎵 bower.json

Client-side packages are listed in the bower.json file. The ASP.NET Core project template pre-configures bower.json with jQuery, jQuery validation, and Bootstrap.

Let's add support for photo albums by installing the Fotorama jQuery plugin. Bower packages can be installed either via the Manage Bower Packages UI or manually in the bower.json file.

Installation via Manage Bower Packages UI

1. Right-click the project name in Solution Explorer, and select the "Manage Bower Packages" menu option.

2. In the window that appears, click the "Browse" tab, and filter the packages list by typing "fotorama" into the search box:
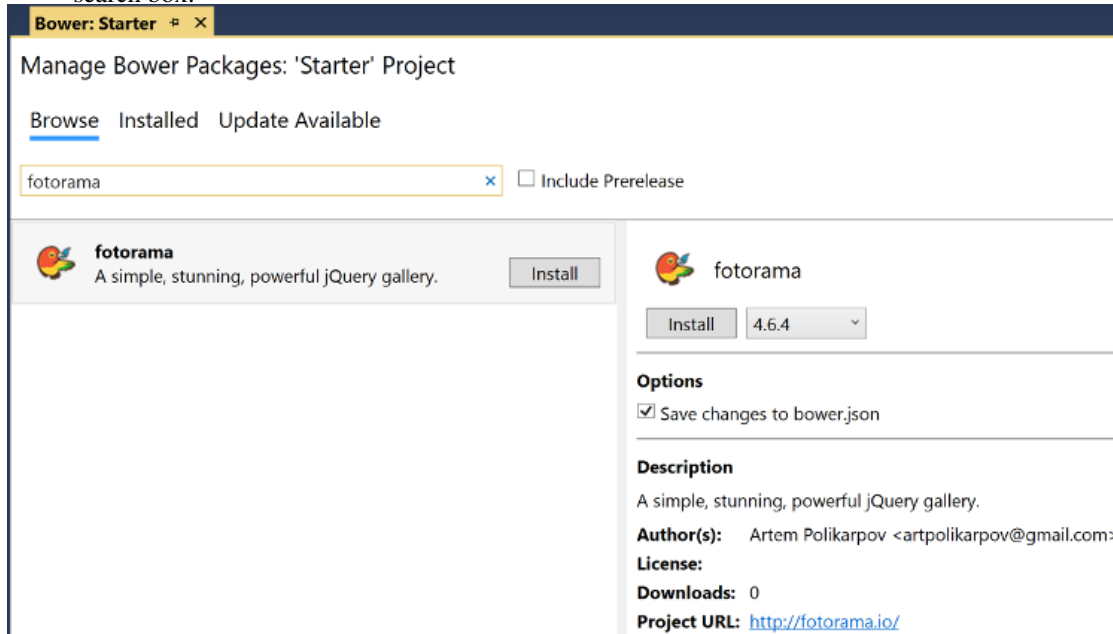
1. Confirm that the "Save changes to bower.json" checkbox is checked, select the desired version from the drop-down list, and click the Install button.

2. Across the bottom status bar of the IDE, an *Installing "fotorama" complete* message appears to indicate a successful installation.

Manual Installation in bower.json

1. At the end of the `dependencies` section in bower.json, add a comma and type "fotorama". Notice as you type that you get IntelliSense with a list of available packages. Select "fotorama" from the list.
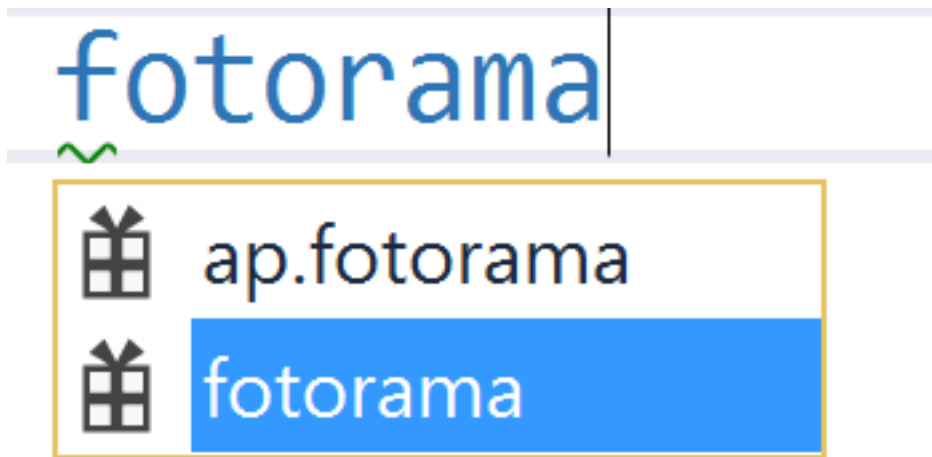
1. Add a colon and then select the latest stable version of the package from the drop-down list. The double quotes will be added automatically.
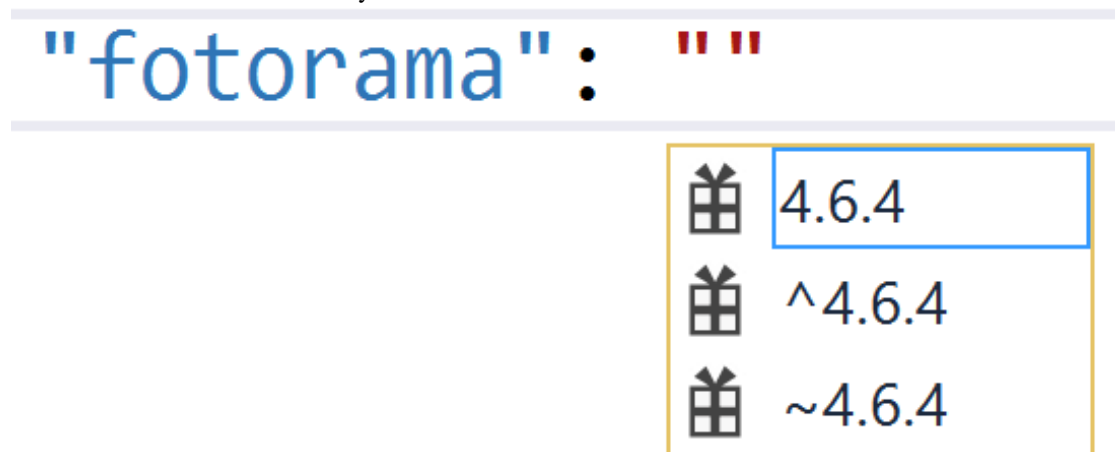
1. Save the bower.json file.

: Visual Studio watches the bower.json file for changes. Upon saving, the *bower install* command is executed. See the Output window's "Bower/npm" view for the exact command which was executed.

Now that the installation step has been completed, expand the twisty to the left of bower.json, and locate the .bowerrc file. Open it, and notice that the `directory` property is set to "wwwroot/lib". This setting indicates the location at which Bower will install the package assets.

```
{
    "directory": "wwwroot/lib"
}
```

In Solution Explorer, expand the *wwwroot* node. The *lib* directory should now contain all of the packages, including the fotorama package.
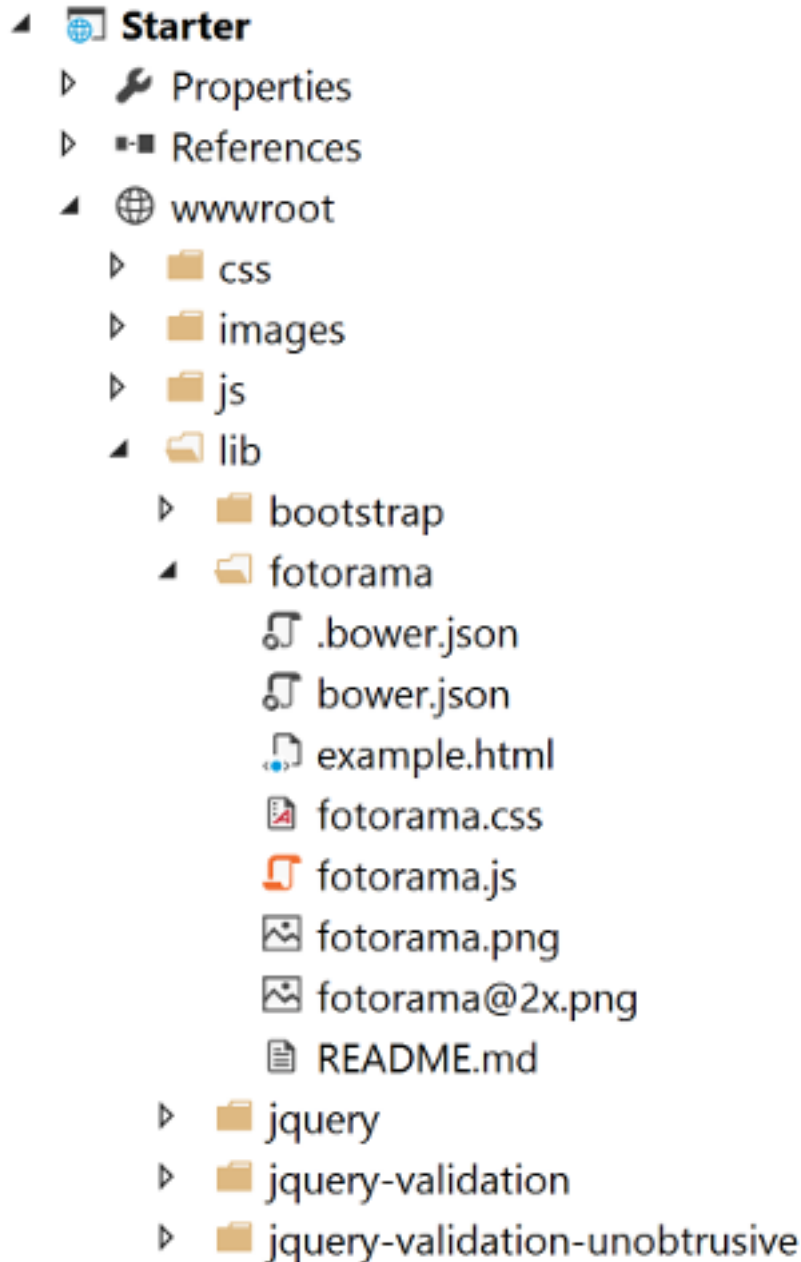
Next, let's add an HTML page to the project. In Solution Explorer, right-click *wwwroot* node and select **Add > New Item > HTML Page**. Name the page Index.html. Replace the contents of the file with the following:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Bower and Fotorama</title>
  <link href="lib/fotorama/fotorama.css" rel="stylesheet" />
</head>
<body>
  <div class="fotorama" data-nav="thumbs">
    <img src="images/asp-net-banners-01.png" />
    <img src="images/asp-net-banners-02.png" />
```

```html
    <img src="images/banner-01-azure.png" />
    <img src="images/banner-02-vs.png" />
  </div>
  <script src="lib/jquery/dist/jquery.js"></script>
  <script src="lib/fotorama/fotorama.js"></script>
</body>
</html>
```

This example uses images currently available inside *wwwroot/images*, but you can add any images on hand.

Press `Ctrl+Shift+W` to display the page in the browser. The control displays the images and allows navigation by clicking the thumbnail list below the main image. This quick test shows that Bower installed the correct packages and dependencies.
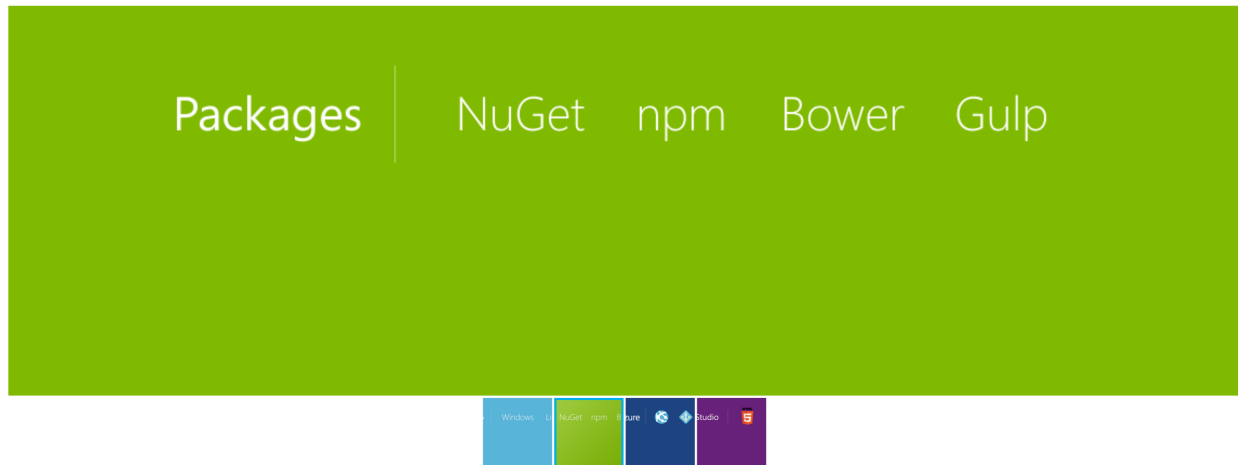
Exploring the Client Build Process

Most ASP.NET Core project templates are already configured to use Bower. This next walkthrough starts with an empty ASP.NET Core project and adds each piece manually, so you can get a feel for how Bower is used in a project. See what happens to the project structure and the runtime output as each configuration change is made to the project.

The general steps to use the client-side build process with Bower are:

  • Define and download packages used in your project.

  • Reference packages from your web pages.

Define Packages

The first step is to define the packages your application needs and to download them. This example uses Bower to load jQuery and Bootstrap in the desired location.

1. In Visual Studio, create a new ASP.NET Web Application.

2. In the **New ASP.NET Project** dialog, select the ASP.NET Core **Empty** project template and click **OK**.

3. In Solution Explorer, the *src* directory includes a project.json file, and *wwwroot* and *Dependencies* nodes. The project directory will look like the screenshot below.
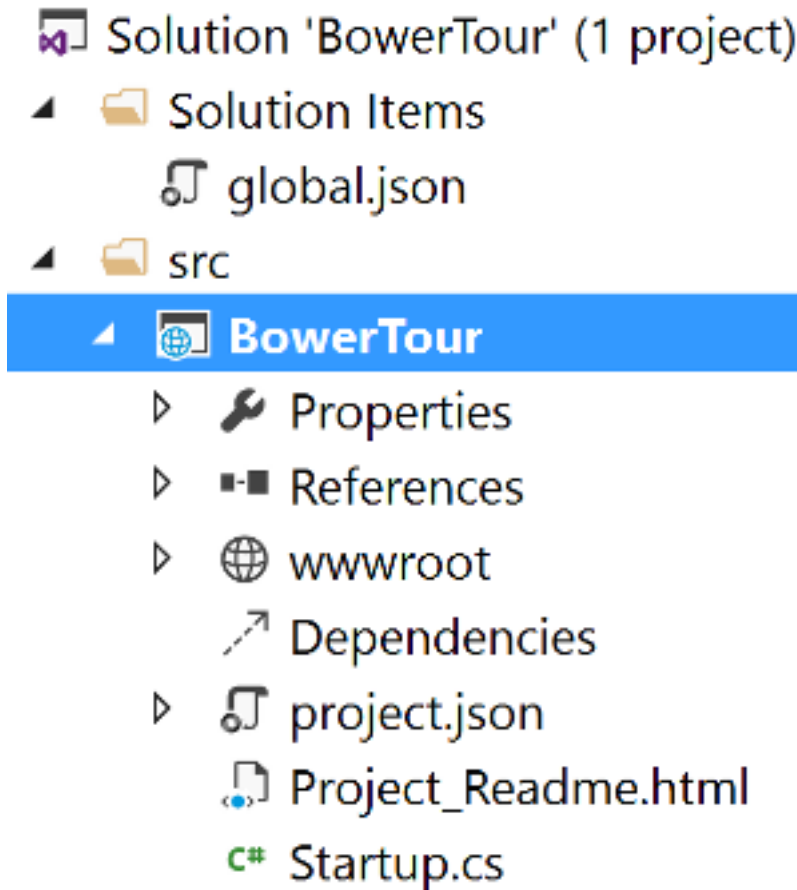
1. In Solution Explorer, right-click the project, and add the following item:

   - Bower Configuration File – bower.json

: The Bower Configuration File item template also adds a .bowerrc file.

---

1. Open bower.json, and add jquery and bootstrap to the `dependencies` section. As an alternative to the manual file editing, the "Manage Bower Packages" UI may be used. The resulting bower.json file should look like the example here. The versions will change over time, so use the latest stable build version from the drop-down list.

```
{
  "name": "ASP.NET",
  "private": true,
  "dependencies": {
    "jquery": "2.1.4",
    "bootstrap": "3.3.5"
  }
}
```

1. Save the bower.json file.

The project should now include *bootstrap* and *jQuery* directories in two locations: *Dependencies/Bower* and *wwwroot/lib*. It's the .bowerrc file which instructed Bower to install the assets within *wwwroot/lib*.
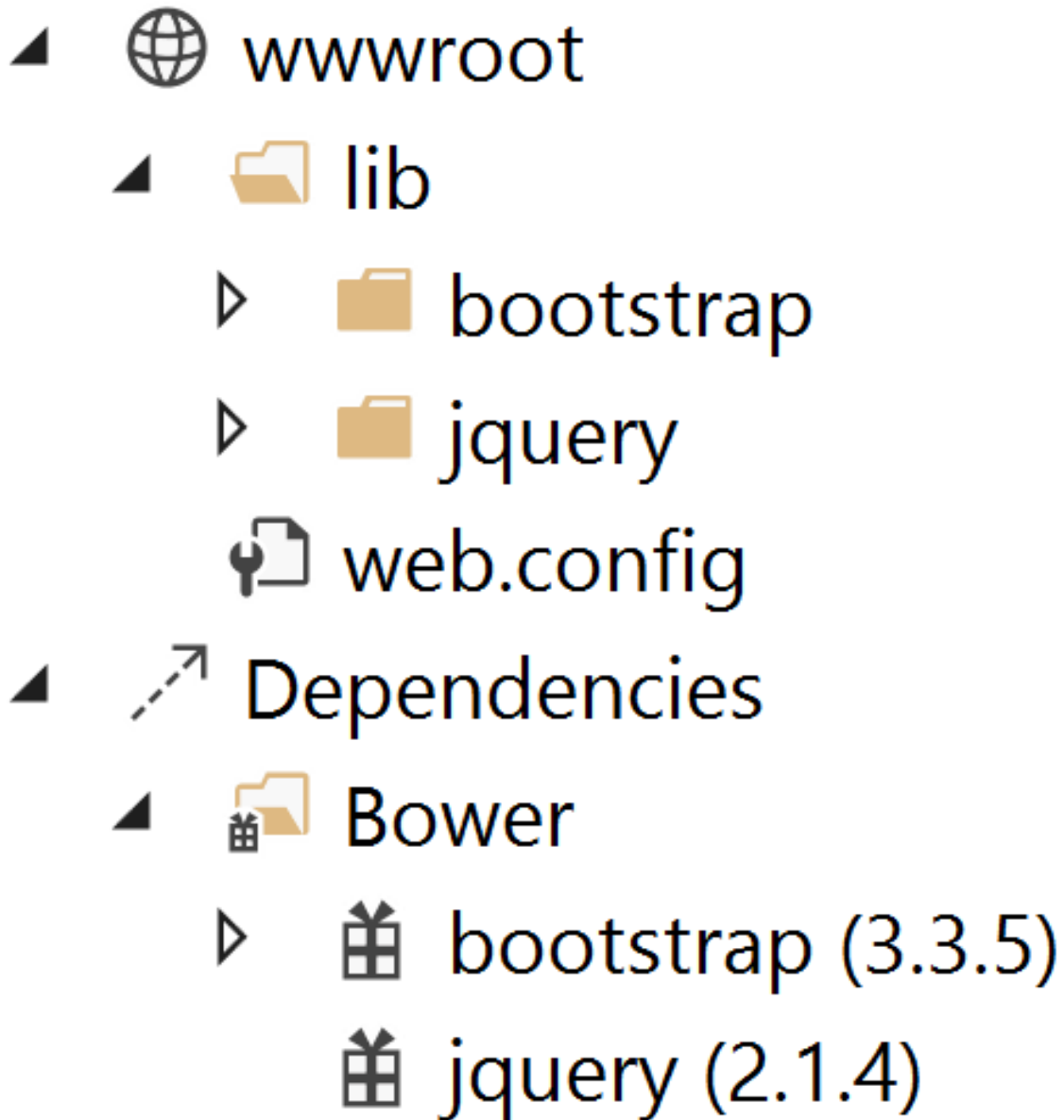
Reference Packages

Now that Bower has copied the client support packages needed by the application, you can test that an HTML page can use the deployed jQuery and Bootstrap functionality.

1. Right-click *wwwroot* and select **Add > New Item > HTML Page**. Name the page Index.html.

2. Add the CSS and JavaScript references.

- In Solution Explorer, expand *wwwroot/lib/bootstrap* and locate bootstrap.css. Drag this file into the `head` element of the HTML page.

- Drag jquery.js and bootstrap.js to the end of the `body` element.

Make sure bootstrap.js follows jquery.js, so that jQuery is loaded first.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Bower Example</title>
  <link href="lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body>

  <script src="lib/jquery/dist/jquery.js"></script>
  <script src="lib/bootstrap/dist/js/bootstrap.js"></script>
</body>
</html>
```

Use the Installed Packages

Add jQuery and Bootstrap components to the page to verify that the web application is configured correctly.

1. Inside the `body` tag, above the `script` references, add a `div` element with the Bootstrap **jumbotron** class and an anchor tag.

```
<div class="jumbotron">
  <h1>Using the jumbotron style</h1>
  <p><a class="btn btn-primary btn-lg" role="button">
    Stateful button</a></p>
</div>
```

1. Add the following code after the jQuery and Bootstrap `script` references.

```
<script>
  $(".btn").click(function() {
    $(this).text('loading')
      .delay(1000)
      .queue(function () {
        $(this).text('reset');
        $(this).dequeue();
      });
  });
</script>
```

1. Within the `Configure` method of the Startup.cs file, add a call to the `UseStaticFiles` extension method. This middleware adds files, found within the web root, to the request pipeline. This line of code will look as follows:

```
app.UseStaticFiles();
```

: Be sure to install the `Microsoft.AspNetCore.StaticFiles` NuGet package. Without it, the `UseStaticFiles` extension method will not resolve.

---

1. With the Index.html file opened, press `Ctrl+Shift+W` to view the page in the browser. Verify that the jumbotron styling is applied, the jQuery code responds when the button is clicked, and that the Bootstrap button changes state.

1.8.4 Building Beautiful, Responsive Sites with Bootstrap

By Steve Smith

Bootstrap is currently the most popular web framework for developing responsive web applications. It offers a number of features and benefits that can improve your users' experience with your web site, whether you're a novice at front-end design and development or an expert. Bootstrap is deployed as a set of CSS and JavaScript files, and is designed to help your website or application scale efficiently from phones to tablets to desktops.

Sections:

- *Getting Started*
- *Basic Templates and Features*
- *More Themes*
- *Components*
- *JavaScript Support*
- *Summary*

Getting Started

There are several ways to get started with Bootstrap. If you're starting a new web application in Visual Studio, you can choose the default starter template for ASP.NET Core, in which case Bootstrap will come pre-installed:
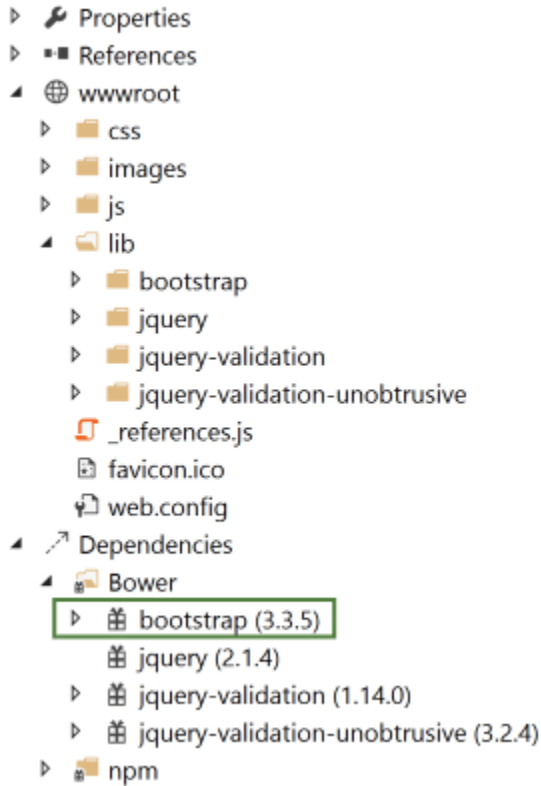
Adding Bootstrap to an ASP.NET Core project is simply a matter of adding it to `bower.json` as a dependency:

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6",
    "jquery": "2.2.0",
    "jquery-validation": "1.14.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

This is the recommended way to add Bootstrap to an ASP.NET Core project.

You can also install bootstrap using one of several package managers, such as bower, npm, or NuGet. In each case, the process is essentially the same:

Bower

```
bower install bootstrap
```

npm

```
npm install bootstrap
```

NuGet

```
Install-Package bootstrap
```

: The recommended way to install client-side dependencies like Bootstrap in ASP.NET Core is via Bower (using `bower.json`, as shown above). The use of npm/NuGet are shown to demonstrate how easily Bootstrap can be added to other kinds of web applications, including earlier versions of ASP.NET.

If you're referencing your own local versions of Bootstrap, you'll need to reference them in any pages that will use it. In production you should reference bootstrap using a CDN. In the default ASP.NET site template, the `_Layout.cshtml` file does so like this:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebApplication1</title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.mi
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-te
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</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=".nav
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">WebAppli
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
                </ul>
                @await Html.PartialAsync("_LoginPartial")
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
```

```
        <footer>
            <p>&copy; 2016 - WebApplication1</p>
        </footer>
    </div>

    <environment names="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment names="Staging,Production">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery">
        </script>
        <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("scripts", required: false)
</body>
</html>
```

: If you're going to be using any of Bootstrap's jQuery plugins, you will also need to reference jQuery.

Basic Templates and Features

The most basic Bootstrap template looks very much like the _Layout.cshtml file shown above, and simply includes a basic menu for navigation and a place to render the rest of the page.

Basic Navigation

The default template uses a set of `<div>` elements to render a top navbar and the main body of the page. If you're using HTML5, you can replace the first `<div>` tag with a `<nav>` tag to get the same effect, but with more precise semantics. Within this first `<div>` you can see there are several others. First, a `<div>` with a class of "container", and then within that, two more `<div>` elements: "navbar-header" and "navbar-collapse". The navbar-header div includes a button that will appear when the screen is below a certain minimum width, showing 3 horizontal lines (a so-called "hamburger icon"). The icon is rendered using pure HTML and CSS; no image is required. This is the code that displays the icon, with each of the <span> tags rendering one of the white bars:

```
<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>
```

It also includes the application name, which appears in the top left. The main navigation menu is rendered by the `<ul>` element within the second div, and includes links to Home, About, and Contact. Additional links for Register and Login are added by the _LoginPartial line on line 29. Below the navigation, the main body of each page is rendered in another `<div>`, marked with the "container" and "body-content" classes. In the simple default _Layout

file shown here, the contents of the page are rendered by the specific View associated with the page, and then a simple `<footer>` is added to the end of the `<div>` element. You can see how the built-in About page appears using this template:
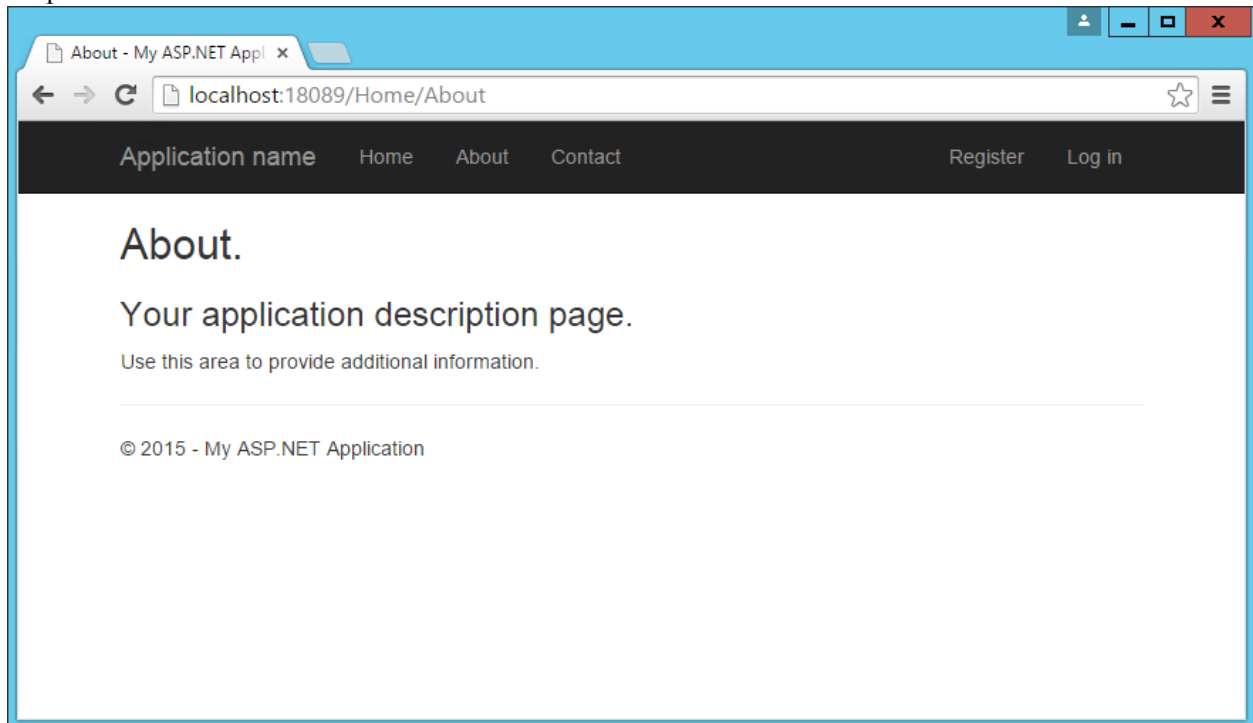
The collapsed navbar, with "hamburger" button in the top right, appears when the window drops below a certain width:
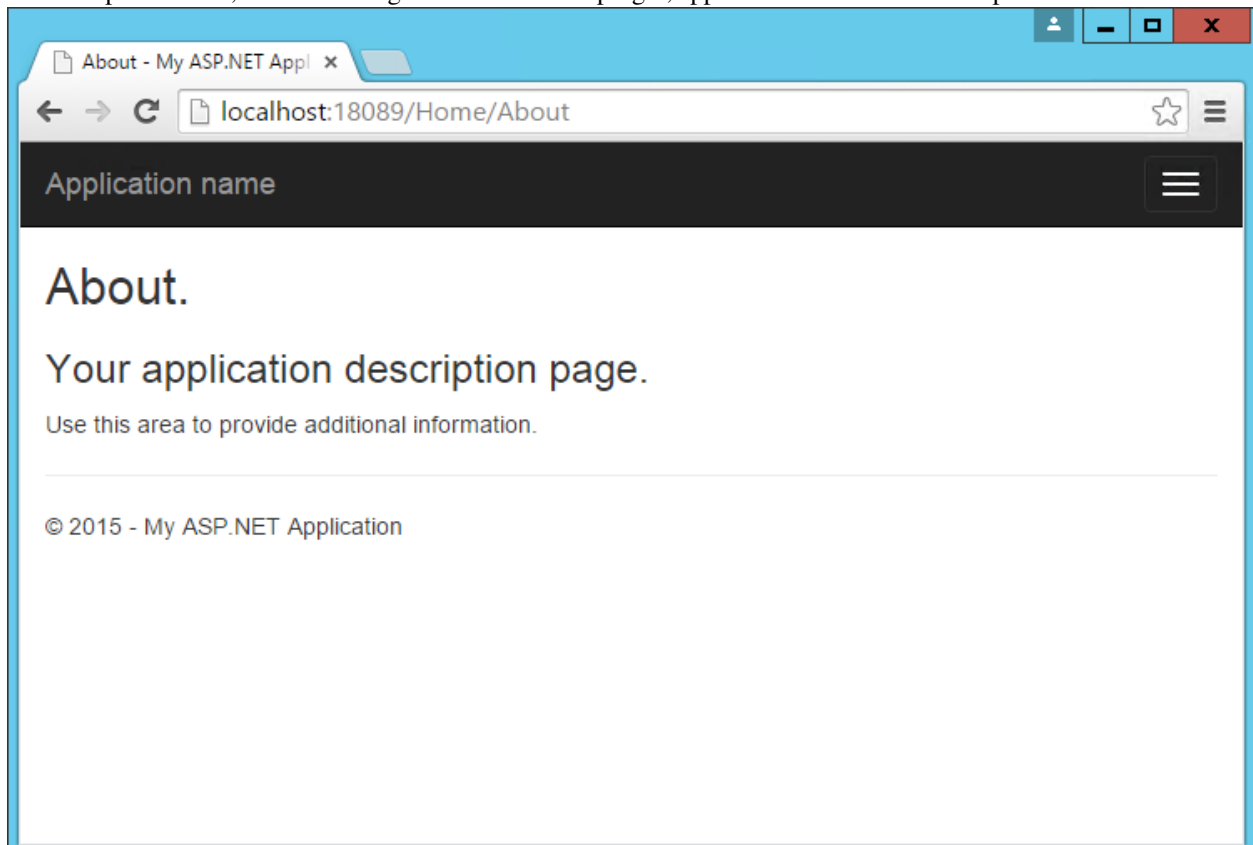
Clicking the icon reveals the menu items in a vertical drawer that slides down from the top of the page:
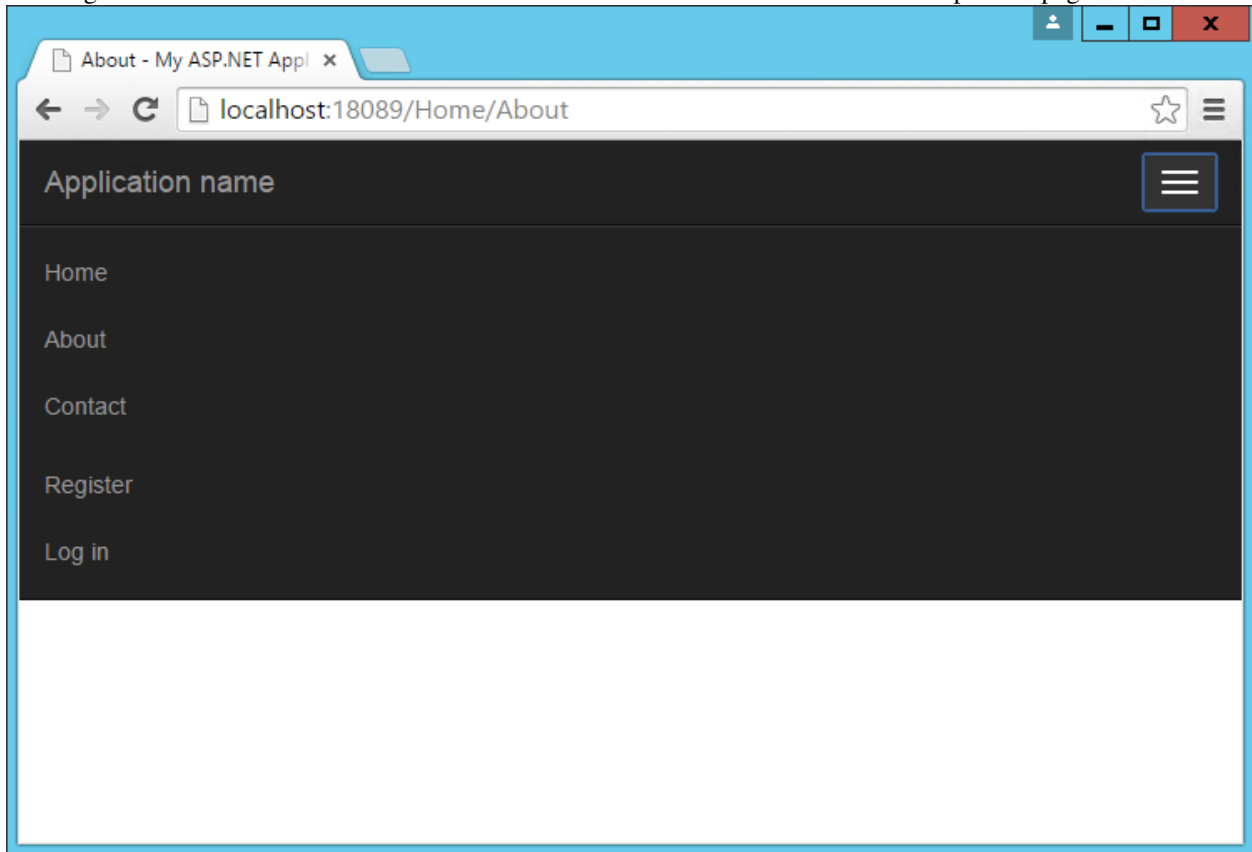
Typography and Links

Bootstrap sets up the site's basic typography, colors, and link formatting in its CSS file. This CSS file includes default styles for tables, buttons, form elements, images, and more (learn more). One particularly useful feature is the grid layout system, covered next.

Grids

One of the most popular features of Bootstrap is its grid layout system. Modern web applications should avoid using the `<table>` tag for layout, instead restricting the use of this element to actual tabular data. Instead, columns and rows can be laid out using a series of `<div>` elements and the appropriate CSS classes. There are several advantages to this approach, including the ability to adjust the layout of grids to display vertically on narrow screens, such as on phones.

Bootstrap's grid layout system is based on twelve columns. This number was chosen because it can be divided evenly into 1, 2, 3, or 4 columns, and column widths can vary to within 1/12th of the vertical width of the screen. To start using the grid layout system, you should begin with a container `<div>` and then add a row `<div>`, as shown here:

```
<div class="container">
  <div class="row">

  </div>
</div>
```

Next, add additional `<div>` elements for each column, and specify the number of columns that `<div>` should occupy (out of 12) as part of a CSS class starting with "col-md-". For instance, if you want to simply have two columns of equal size, you would use a class of "col-md-6" for each one. In this case "md" is short for "medium" and refers to standard-sized desktop computer display sizes. There are four different options you can choose from, and each will be used for higher widths unless overridden (so if you want the layout to be fixed regardless of screen width, you can just specify xs classes).

CSS Class Prefix	Device Tier	Width
col-xs-	Phones	< 768px
col-sm-	Tablets	>= 768px
col-md-	Desktops	>= 992px
col-lg-	Larger Desktop Displays	>= 1200px

When specifying two columns both with "col-md-6" the resulting layout will be two columns at desktop resolutions, but these two columns will stack vertically when rendered on smaller devices (or a narrower browser window on a desktop), allowing users to easily view content without the need to scroll horizontally.

Bootstrap will always default to a single-column layout, so you only need to specify columns when you want more than one column. The only time you would want to explicitly specify that a `<div>` take up all 12 columns would be to override the behavior of a larger device tier. When specifying multiple device tier classes, you may need to reset the column rendering at certain points. Adding a clearfix div that is only visible within a certain viewport can achieve this, as shown here:

```
<p>Use this area to provide additional information.</p>
<style>
    [class*="col-"] {
        background-color: lightblue;
        border: 1px solid blue;
    }
</style>
<div class="container">
    <div class="row">
        <div class="col-xs-12 col-md-8">
            One
        </div>
        <div class="col-xs-6 col-md-4">
            Two
        </div>
        <div class="col-xs-6 col-md-4">
            Three
        </div>
        <div class="clearfix visible-xs"></div>
        <div class="col-xs-12 col-md-4">
            Four
        </div>
        <div class="col-xs-12 col-md-4">
            Five
        </div>
    </div>
</div>
```


In the above example, One and Two share a row in the "md" layout, while Two and Three share a row in the "xs" layout. Without the clearfix `<div>`, Two and Three are not shown correctly in the "xs" view (note that only One, Four, and Five are shown):

In this example, only a single row `<div>` was used, and Bootstrap still mostly did the right thing with regard to the layout and stacking of the columns. Typically, you should specify a row `<div>` for each horizontal row your layout requires, and of course you can nest Bootstrap grids within one another. When you do, each nested grid will occupy 100% of the width of the element in which it is placed, which can then be subdivided using column classes.

Jumbotron

If you've used the default ASP.NET MVC templates in Visual Studio 2012 or 2013, you've probably seen the Jumbotron in action. It refers to a large full-width section of a page that can be used to display a large background image, a call to action, a rotator, or similar elements. To add a jumbotron to a page, simply add a `<div>` and give it a class of "jumbotron", then place a container `<div>` inside and add your content. We can easily adjust the standard About page to use a jumbotron for the main headings it displays:

```
<style>
    .jumbotron {
        background-color: lightblue;
    }
</style>

<div class="jumbotron">
    <div class="container">
        <h2>@ViewBag.Title.</h2>
        <h3>@ViewBag.Message</h3>
    </div>
</div>
<p>Use this area to provide additional information.</p>
```

Buttons

The default button classes and their colors are shown in the figure below.

```
<h2>Theme Options</h2>
<p>
    <button type="button" class="btn btn-default">btn-default</button>
    <button type="button" class="btn btn-primary">btn-primary</button>
    <button type="button" class="btn btn-success">btn-success</button>
    <button type="button" class="btn btn-info">btn-info</button>
    <button type="button" class="btn btn-warning">btn-warning</button>
    <button type="button" class="btn btn-danger">btn-danger</button>
    <button type="button" class="btn btn-link">btn-link</button>
</p>
```

Badges

Badges refer to small, usually numeric callouts next to a navigation item. They can indicate a number of messages or notifications waiting, or the presence of updates. Specifying such badges is as simple as adding a <span> containing the text, with a class of "badge":

```
<h3>Badges</h3>
<p>
    Inbox <span class="badge">27</span>
</p>
```


Alerts

You may need to display some kind of notification, alert, or error message to your application's users. That's where the standard alert classes come in. There are four different severity levels, with associated color schemes:

```
<h3>Alerts</h3>
<div class="alert alert-success">
    <strong>Success!</strong> Well done.
</div>
<div class="alert alert-info">
    <strong>FYI</strong> You might need to know this.
</div>
<div class="alert alert-warning">
    <strong>Warning!</strong> Use caution.
</div>
<div class="alert alert-danger">
    <strong>Danger!</strong> Something's wrong!
</div>
```


Navbars and Menus

Our layout already includes a standard navbar, but the Bootstrap theme supports additional styling options. We can also easily opt to display the navbar vertically rather than horizontally if that's preferred, as well as adding sub-navigation items in flyout menus. Simple navigation menus, like tab strips, are built on top of <ul> elements. These can be created very simply by just providing them with the CSS classes "nav" and "nav-tabs":

```
<h3>Nav and Menus</h3>
<p>Tabs:</p>
<ul class="nav nav-tabs">
    <li class="active"><a href="#">One</a></li>
    <li><a href="#">Two</a></li>
    <li><a href="#">Three</a></li>
</ul>
```


Navbars are built similarly, but are a bit more complex. They start with a `<nav>` or `<div>` with a class of "navbar", within which a container div holds the rest of the elements. Our page includes a navbar in its header already – the one shown below simply expands on this, adding support for a dropdown menu:

```html
<p>Navbars:</p>
<div class="navbar navbar-default">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="collapse"
                    data-target=".navbar-collapse">
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">Menu</a>
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li class="active"><a href="#">One</a></li>
                <li><a href="#two">Two</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown"
                       role="button" aria-expanded="false">Choose <span class="caret"></
                    <ul class="dropdown-menu" role="menu">
                        <li><a href="#">Alpha</a></li>
                        <li><a href="#">Bravo</a></li>
                        <li class="divider"></li>
                        <li class="dropdown-header">Subheader</li>
                        <li><a href="#">Charlie</a></li>
                        <li><a href="#">Delta</a></li>
                    </ul>
                </li>
            </ul>
        </div>
    </div>
</div>
```

Additional Elements

The default theme can also be used to present HTML tables in a nicely formatted style, including support for striped views. There are labels with styles that are similar to those of the buttons. You can create custom Dropdown menus that support additional styling options beyond the standard HTML `<select>` element, along with Navbars like the one our default starter site is already using. If you need a progress bar, there are several styles to choose from, as well as List Groups and panels that include a title and content. Explore additional options within the standard Bootstrap Theme here:

http://getbootstrap.com/examples/theme/

More Themes

You can extend the standard Bootstrap Theme by overriding some or all of its CSS, adjusting the colors and styles to suit your own application's needs. If you'd like to start from a ready-made theme, there are several theme galleries available online that specialize in Bootstrap Themes, such as WrapBootstrap.com (which has a variety of commercial themes) and Bootswatch.com (which offers free themes). Some of the paid templates available provide a great deal of functionality on top of the basic Bootstrap theme, such as rich support for administrative menus, and dashboards with rich charts and gauges. An example of a popular paid template is Inspinia, currently for sale for $18, which includes an ASP.NET MVC5 template in addition to AngularJS and static HTML versions. A sample screenshot is shown below.

If you're interested in building your own dashboard, you may wish to start from the free example available here: http://getbootstrap.com/examples/dashboard/.

Components

In addition to those elements already discussed, Bootstrap includes support for a variety of built-in UI components.

Glyphicons

Bootstrap includes icon sets from Glyphicons (http://glyphicons.com), with over 200 icons freely available for use within your Bootstrap-enabled web application. Here's just a small sample:

Input Groups

Input groups allow bundling of additional text or buttons with an input element, providing the user with a more intuitive experience:

Breadcrumbs

Breadcrumbs are a common UI component used to show a user their recent history or depth within a site's navigation hierarchy. Add them easily by applying the "breadcrumb" class to any `<ol>` list element. Include built-in support for pagination by using the "pagination" class on a `<ul>` element within a `<nav>`. Add responsive embedded slideshows and video by using `<iframe>`, `<embed>`, `<video>`, or `<object>` elements, which Bootstrap will style automatically. Specify a particular aspect ratio by using specific classes like "embed-responsive-16by9".

JavaScript Support

Bootstrap's JavaScript library includes API support for the included components, allowing you to control their behavior programmatically within your application. In addition, bootstrap.js includes over a dozen custom jQuery plugins, providing additional features like transitions, modal dialogs, scroll detection (updating styles based on where the user

has scrolled in the document), collapse behavior, carousels, and affixing menus to the window so they do not scroll off the screen. There's not sufficient room to cover all of the JavaScript add-ons built into Bootstrap – to learn more please visit http://getbootstrap.com/javascript/.

Summary

Bootstrap provides a web framework that can be used to quickly and productively lay out and style a wide variety of websites and applications. Its basic typography and styles provide a pleasant look and feel that can easily be manipulated through custom theme support, which can be hand-crafted or purchased commercially. It supports a host of web components that in the past would have required expensive third-party controls to accomplish, while supporting modern and open web standards.

1.8.5 Knockout.js MVVM Framework

By Steve Smith

Knockout is a popular JavaScript library that simplifies the creation of complex data-based user interfaces. It can be used alone or with other libraries, such as jQuery. Its primary purpose is to bind UI elements to an underlying data model defined as a JavaScript object, such that when changes are made to the UI, the model is updated, and vice versa. Knockout facilitates the use of a Model-View-ViewModel (MVVM) pattern in a web application's client-side behavior. The two main concepts one must learn when working with Knockout's MVVM implementation are Observables and Bindings.

Sections:

- *Getting Started with Knockout in ASP.NET Core*
- *Observables, ViewModels, and Simple Binding*
- *Control Flow*
- *Templates*
- *Components*
- *Communicating with APIs*
- *Summary*

Getting Started with Knockout in ASP.NET Core

Knockout is deployed as a single JavaScript file, so installing and using it is very straightforward using *bower*. Assuming you already have *bower* and *gulp* configured, open bower.json in your ASP.NET Core project and add the knockout dependency as shown here:

```
{
  "name": "KnockoutDemo",
  "private": true,
  "dependencies": {
    "knockout" : "^3.3.0"
  },
  "exportsOverride": {
  }
}
```

With this in place, you can then manually run bower by opening the Task Runner Explorer (under *View → Other Windows → Task Runner Explorer*) and then under Tasks, right-click on bower and select Run. The result should appear similar to this:

Now if you look in your project's `wwwroot` folder, you should see knockout installed under the lib folder.

It's recommended that in your production environment you reference knockout via a Content Delivery Network, or CDN, as this increases the likelihood that your users will already have a cached copy of the file and thus will not need to download it at all. Knockout is available on several CDNs, including the Microsoft Ajax CDN, here:

http://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js

To include Knockout on a page that will use it, simply add a `<script>` element referencing the file from wherever you will be hosting it (with your application, or via a CDN):

```
<script type="text/javascript" src="knockout-3.3.0.js"></script>
```

Observables, ViewModels, and Simple Binding

You may already be familiar with using JavaScript to manipulate elements on a web page, either via direct access to the DOM or using a library like jQuery. Typically this kind of behavior is achieved by writing code to directly set element values in response to certain user actions. With Knockout, a declarative approach is taken instead, through which

elements on the page are bound to properties on an object. Instead of writing code to manipulate DOM elements, user actions simply interact with the ViewModel object, and Knockout takes care of ensuring the page elements are synchronized.

As a simple example, consider the page list below. It includes a `<span>` element with a `data-bind` attribute indicating that the text content should be bound to authorName. Next, in a JavaScript block a variable viewModel is defined with a single property, `authorName`, set to some value. Finally, a call to `ko.applyBindings` is made, passing in this viewModel variable.

```html
1  <html>
2    <head>
3      <script type="text/javascript" src="lib/knockout/knockout.js"></script>
4    </head>
5    <body>
6      <h1>Some Article</h1>
7      <p>
8        By <span data-bind="text: authorName"></span>
9      </p>
10     <script type="text/javascript">
11       var viewModel = {
12         authorName: 'Steve Smith'
13       };
14       ko.applyBindings(viewModel);
15     </script>
16   </body>
17 </html>
```

When viewed in the browser, the content of the <span> element is replaced with the value in the viewModel variable:

We now have simple one-way binding working. Notice that nowhere in the code did we write JavaScript to assign a value to the span's contents. If we want to manipulate the ViewModel, we can take this a step further and add an HTML input textbox, and bind to its value, like so:

```
<p>
  Author Name: <input type="text" data-bind="value: authorName" />
</p>
```

Reloading the page, we see that this value is indeed bound to the input box:

However, if we change the value in the textbox, the corresponding value in the `<span>` element doesn't change. Why not?

The issue is that nothing notified the `<span>` that it needed to be updated. Simply updating the ViewModel isn't by itself sufficient, unless the ViewModel's properties are wrapped in a special type. We need to use **observables** in the ViewModel for any properties that need to have changes automatically updated as they occur. By changing the ViewModel to use `ko.observable("value")` instead of just "value", the ViewModel will update any HTML elements that are bound to its value whenever a change occurs. Note that input boxes don't update their value until they lose focus, so you won't see changes to bound elements as you type.

: Adding support for live updating after each keypress is simply a matter of adding `valueUpdate: "afterkeydown"` to the `data-bind` attribute's contents.

Our viewModel, after updating it to use ko.observable:

```
var viewModel = {
  authorName: ko.observable('Steve Smith')
};
ko.applyBindings(viewModel);
```

Knockout supports a number of different kinds of bindings. So far we've seen how to bind to `text` and to `value`. You can also bind to any given attribute. For instance, to create a hyperlink with an anchor tag, the `src` attribute can be bound to the viewModel. Knockout also supports binding to functions. To demonstrate this, let's update the viewModel to include the author's twitter handle, and display the twitter handle as a link to the author's twitter page. We'll do this in three stages.

First, add the HTML to display the hyperlink, which we'll show in parentheses after the author's name:

```
<h1>Some Article</h1>
<p>
  By <span data-bind="text: authorName"></span>
  (<a data-bind="attr: { href: twitterUrl}, text: twitterAlias" ></a>)
</p>
```

Next, update the viewModel to include the twitterUrl and twitterAlias properties:

```
var viewModel = {
  authorName: ko.observable('Steve Smith'),
  twitterAlias: ko.observable('@ardalis'),
  twitterUrl: ko.computed(function() {
    return "https://twitter.com/";
  }, this)
};
ko.applyBindings(viewModel);
```

Notice that at this point we haven't yet updated the twitterUrl to go to the correct URL for this twitter alias – it's just pointing at twitter.com. Also notice that we're using a new Knockout function, computed, for twitterUrl. This is an observable function that will notify any UI elements if it changes. However, for it to have access to other properties in the viewModel, we need to change how we are creating the viewModel, so that each property is its own statement.

The revised viewModel declaration is shown below. It is now declared as a function. Notice that each property is its own statement now, ending with a semicolon. Also notice that to access the twitterAlias property value, we need to execute it, so its reference includes ().

```
function viewModel() {
  this.authorName = ko.observable('Steve Smith');
  this.twitterAlias = ko.observable('@ardalis');

  this.twitterUrl = ko.computed(function() {
    return "https://twitter.com/" + this.twitterAlias().replace('@','');
  }, this)
};
ko.applyBindings(viewModel);
```

The result works as expected in the browser:

Knockout also supports binding to certain UI element events, such as the click event. This allows you to easily and declaratively bind UI elements to functions within the application's viewModel. As a simple example, we can add a button that, when clicked, modifies the author's twitterAlias to be all caps.

First, we add the button, binding to the button's click event, and referencing the function name we're going to add to the viewModel:

```
<p>
  <button data-bind="click: capitalizeTwitterAlias">Capitalize</button>
</p>
```

Then, add the function to the viewModel, and wire it up to modify the viewModel's state. Notice that to set a new value to the twitterAlias property, we call it as a method and pass in the new value.

```
function viewModel() {
  this.authorName = ko.observable('Steve Smith');
  this.twitterAlias = ko.observable('@ardalis');

  this.twitterUrl = ko.computed(function() {
    return "https://twitter.com/" + this.twitterAlias().replace('@','');
  }, this);

  this.capitalizeTwitterAlias = function() {
    var currentValue = this.twitterAlias();
    this.twitterAlias(currentValue.toUpperCase());
  }
};
ko.applyBindings(viewModel);
```

Running the code and clicking the button modifies the displayed link as expected:

Control Flow

Knockout includes bindings that can perform conditional and looping operations. Looping operations are especially useful for binding lists of data to UI lists, menus, and grids or tables. The foreach binding will iterate over an array. When used with an observable array, it will automatically update the UI elements when items are added or removed from the array, without re-creating every element in the UI tree. The following example uses a new viewModel which includes an observable array of game results. It is bound to a simple table with two columns using a `foreach` binding on the `<tbody>` element. Each `<tr>` element within `<tbody>` will be bound to an element of the gameResults collection.

```
1   <h1>Record</h1>
2   <table>
3     <thead>
4       <tr>
5         <th>Opponent</th>
6         <th>Result</th>
7       </tr>
8     </thead>
9     <tbody data-bind="foreach: gameResults">
10      <tr>
11        <td data-bind="text:opponent"></td>
12        <td data-bind="text:result"></td>
13      </tr>
14    </tbody>
15  </table>
16  <script type="text/javascript">
17    function GameResult(opponent, result) {
18      var self = this;
19      self.opponent = opponent;
20      self.result = ko.observable(result);
21    }
```

```
22
23    function ViewModel() {
24      var self = this;
25
26      self.resultChoices = ["Win", "Loss", "Tie"];
27
28      self.gameResults = ko.observableArray([
29        new GameResult("Brendan", self.resultChoices[0]),
30        new GameResult("Brendan", self.resultChoices[0]),
31        new GameResult("Michelle", self.resultChoices[1])
32      ]);
33    };
34    ko.applyBindings(new ViewModel);
35  </script>
```

Notice that this time we're using ViewModel with a capital "V" because we expect to construct it using "new" (in the applyBindings call). When executed, the page results in the following output:
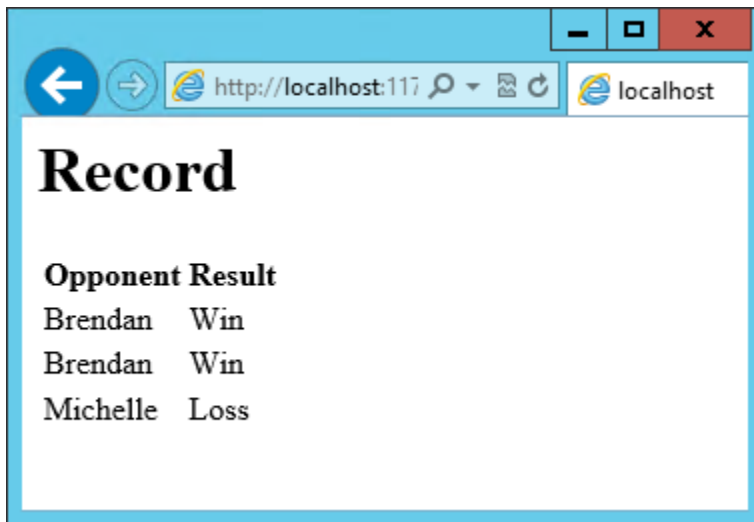
To demonstrate that the observable collection is working, let's add a bit more functionality. We can include the ability to record the results of another game to the ViewModel, and then add a button and some UI to work with this new function. First, let's create the addResult method:

```
// add this to ViewModel()
self.addResult = function() {
  self.gameResults.push(new GameResult("", self.resultChoices[0]));
}
```

Bind this method to a button using the `click` binding:

```
<button data-bind="click: addResult">Add New Result</button>
```

Open the page in the browser and click the button a couple of times, resulting in a new table row with each click:
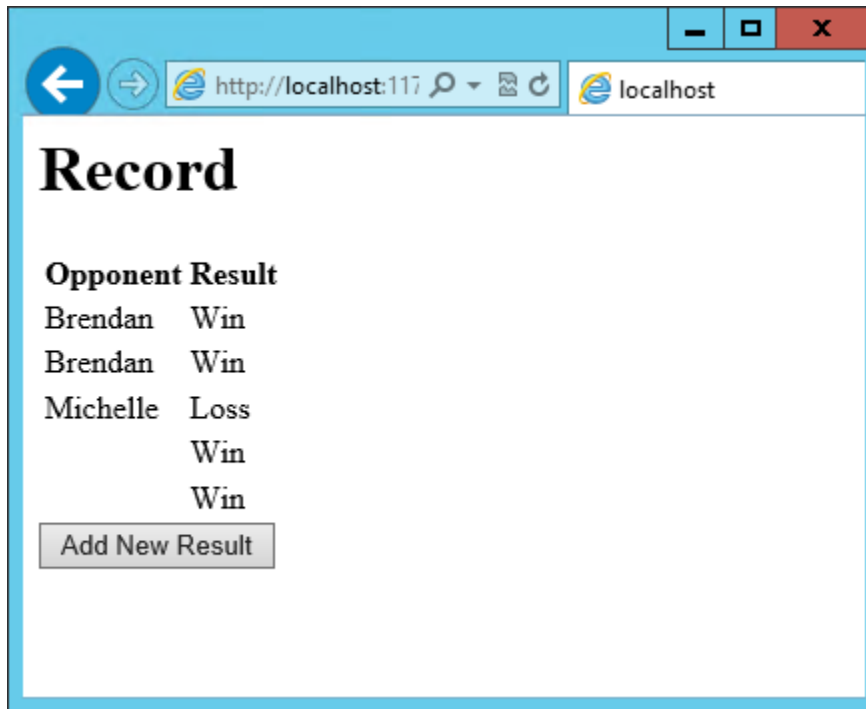
There are a few ways to support adding new records in the UI, typically either inline or in a separate form. We can easily modify the table to use textboxes and dropdownlists so that the whole thing is editable. Just change the `<tr>` element as shown:

```
<tbody data-bind="foreach: gameResults">
  <tr>
    <td><input data-bind="value:opponent" /></td>
    <td><select data-bind="options: $root.resultChoices,
      value:result, optionsText: $data"></select></td>
  </tr>
</tbody>
```

Note that `$root` refers to the root ViewModel, which is where the possible choices are exposed. `$data` refers to whatever the current model is within a given context - in this case it refers to an individual element of the resultChoices array, each of which is a simple string.

With this change, the entire grid becomes editable:
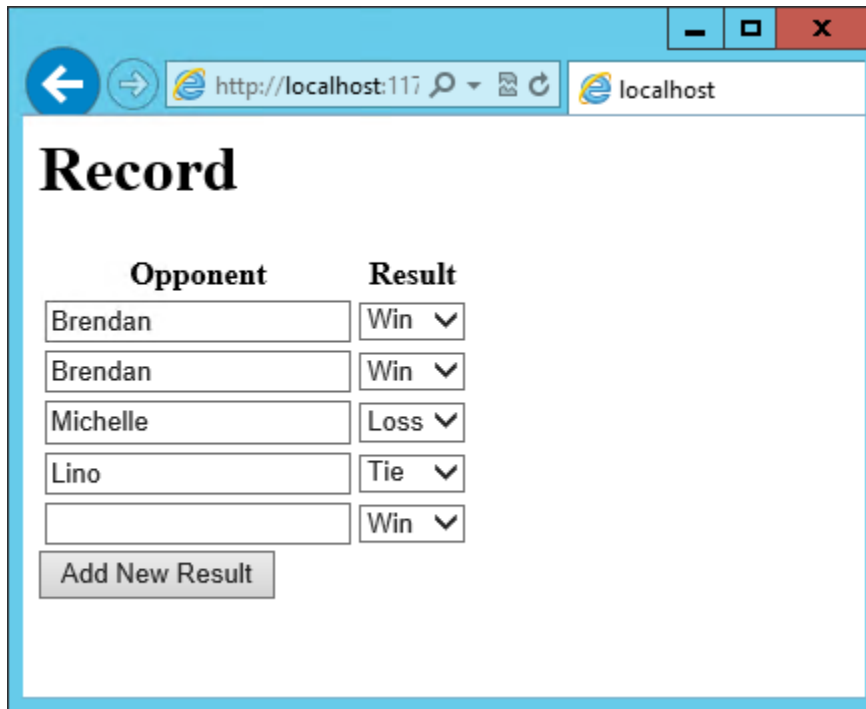
If we weren't using Knockout, we could achieve all of this using jQuery, but most likely it would not be nearly as efficient. Knockout tracks which bound data items in the ViewModel correspond to which UI elements, and only updates those elements that need to be added, removed, or updated. It would take significant effort to achieve this ourselves using jQuery or direct DOM manipulation, and even then if we then wanted to display aggregate results (such as a win-loss record) based on the table's data, we would need to once more loop through it and parse the HTML elements. With Knockout, displaying the win-loss record is trivial. We can perform the calculations within the ViewModel itself, and then display it with a simple text binding and a `<span>`.

To build the win-loss record string, we can use a computed observable. Note that references to observable properties within the ViewModel must be function calls, otherwise they will not retrieve the value of the observable (i.e. `gameResults()` not `gameResults` in the code shown):

```
self.displayRecord = ko.computed(function () {
  var wins = self.gameResults().filter(function (value) { return value.result() == "Win"; }).length;
  var losses = self.gameResults().filter(function (value) { return value.result() == "Loss"; }).lengt
  var ties = self.gameResults().filter(function (value) { return value.result() == "Tie"; }).length;
  return wins + " - " + losses + " - " + ties;
}, this);
```

Bind this function to a span within the `<h1>` element at the top of the page:

```
<h1>Record <span data-bind="text: displayRecord"></span></h1>
```

The result:
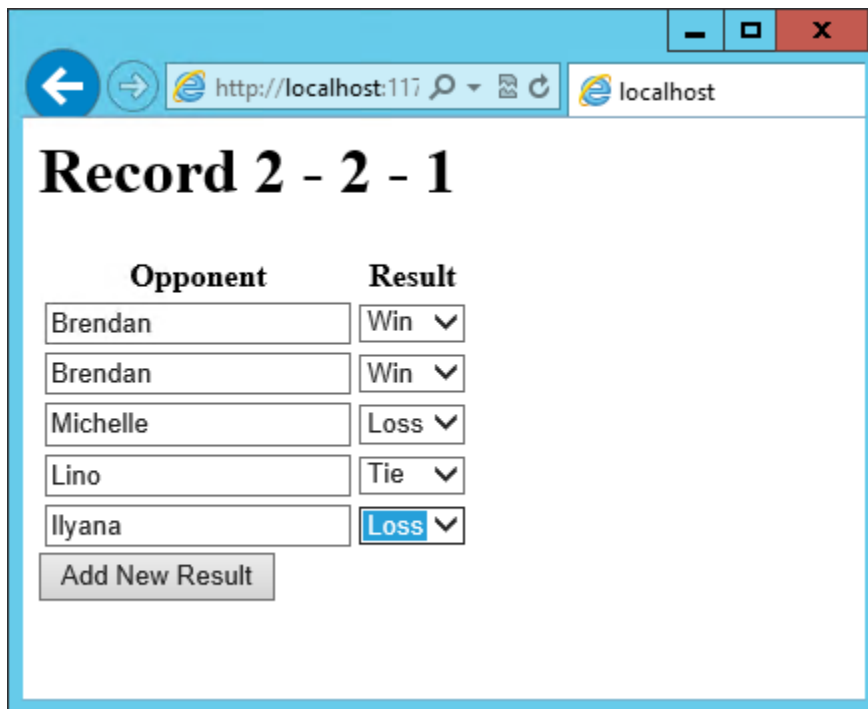
Adding rows or modifying the selected element in any row's Result column will update the record shown at the top of the window.

In addition to binding to values, you can also use almost any legal JavaScript expression within a binding. For example, if a UI element should only appear under certain conditions, such as when a value exceeds a certain threshold, you can specify this logically within the binding expression:

```
<div data-bind="visible: customerValue > 100"></div>
```

This <div> will only be visible when the customerValue is over 100.

Templates

Knockout has support for templates, so that you can easily separate your UI from your behavior, or incrementally load UI elements into a large application on demand. We can update our previous example to make each row its own template by simply pulling the HTML out into a template and specifying the template by name in the data-bind call on <tbody>.

```
<tbody data-bind="template: { name: 'rowTemplate', foreach: gameResults }">
</tbody>
<script type="text/html" id="rowTemplate">
  <tr>
    <td><input data-bind="value:opponent" /></td>
    <td><select data-bind="options: $root.resultChoices,
       value:result, optionsText: $data"></select></td>
  </tr>
</script>
```

Knockout also supports other templating engines, such as the jQuery.tmpl library and Underscore.js's templating engine.

Components

Components allow you to organize and reuse UI code, usually along with the ViewModel data on which the UI code depends. To create a component, you simply need to specify its template and its viewModel, and give it a name. This is done by calling `ko.components.register()`. In addition to defining the templates and viewmodel inline, they can be loaded from external files using a library like require.js, resulting in very clean and efficient code.

Communicating with APIs

Knockout can work with any data in JSON format. A common way to retrieve and save data using Knockout is with jQuery, which supports the `$.getJSON()` function to retrieve data, and the `$.post()` method to send data from the browser to an API endpoint. Of course, if you prefer a different way to send and receive JSON data, Knockout will work with it as well.

Summary

Knockout provides a simple, elegant way to bind UI elements to the current state of the client application, defined in a ViewModel. Knockout's binding syntax uses the data-bind attribute, applied to HTML elements that are to be processed. Knockout is able to efficiently render and update large data sets by tracking UI elements and only processing changes to affected elements. Large applications can break up UI logic using templates and components, which can be loaded on demand from external files. Currently version 3, Knockout is a stable JavaScript library that can improve web applications that require rich client interactivity.

1.8.6 Using Angular for Single Page Applications (SPAs)

By Venkata Koppaka and Scott Addie

In this article, you will learn how to build a SPA-style ASP.NET application using AngularJS.

> **Sections:**
>
> - *What is AngularJS?*
> - *Getting Started*
> - *Key Components*
> - *Angular 2.0*
> - *Summary*
> - *Related Resources*

View or download sample code

What is AngularJS?

AngularJS is a modern JavaScript framework from Google commonly used to work with Single Page Applications (SPAs). AngularJS is open sourced under MIT license, and the development progress of AngularJS can be followed on its GitHub repository. The library is called Angular because HTML uses angular-shaped brackets.

AngularJS is not a DOM manipulation library like jQuery, but it uses a subset of jQuery called jQLite. AngularJS is primarily based on declarative HTML attributes that you can add to your HTML tags. You can try AngularJS in your browser using the Code School website.

Version 1.5.x is the current stable version and the Angular team is working towards a big rewrite of AngularJS for V2.0 which is currently still in development. This article focuses on Angular 1.X with some notes on where Angular is heading with 2.0.

Getting Started

To start using AngularJS in your ASP.NET application, you must either install it as part of your project, or reference it from a content delivery network (CDN).

Installation

There are several ways to add AngularJS to your application. If you're starting a new ASP.NET Core web application in Visual Studio, you can add AngularJS using the built-in *Bower* support. Simply open `bower.json`, and add an entry to the `dependencies` property:

```
1  {
2    "name": "ASP.NET",
3    "private": true,
4    "dependencies": {
5      "bootstrap": "3.3.5",
6      "jquery": "2.1.4",
7      "jquery-validation": "1.14.0",
8      "jquery-validation-unobtrusive": "3.2.4",
9      "angular": "1.4.8",
10     "angular-route": "1.4.8"
11   }
12 }
```

Upon saving the `bower.json` file, Angular will be installed in your project's `wwwroot/lib` folder. Additionally, it will be listed within the `Dependencies/Bower` folder. See the screenshot below.
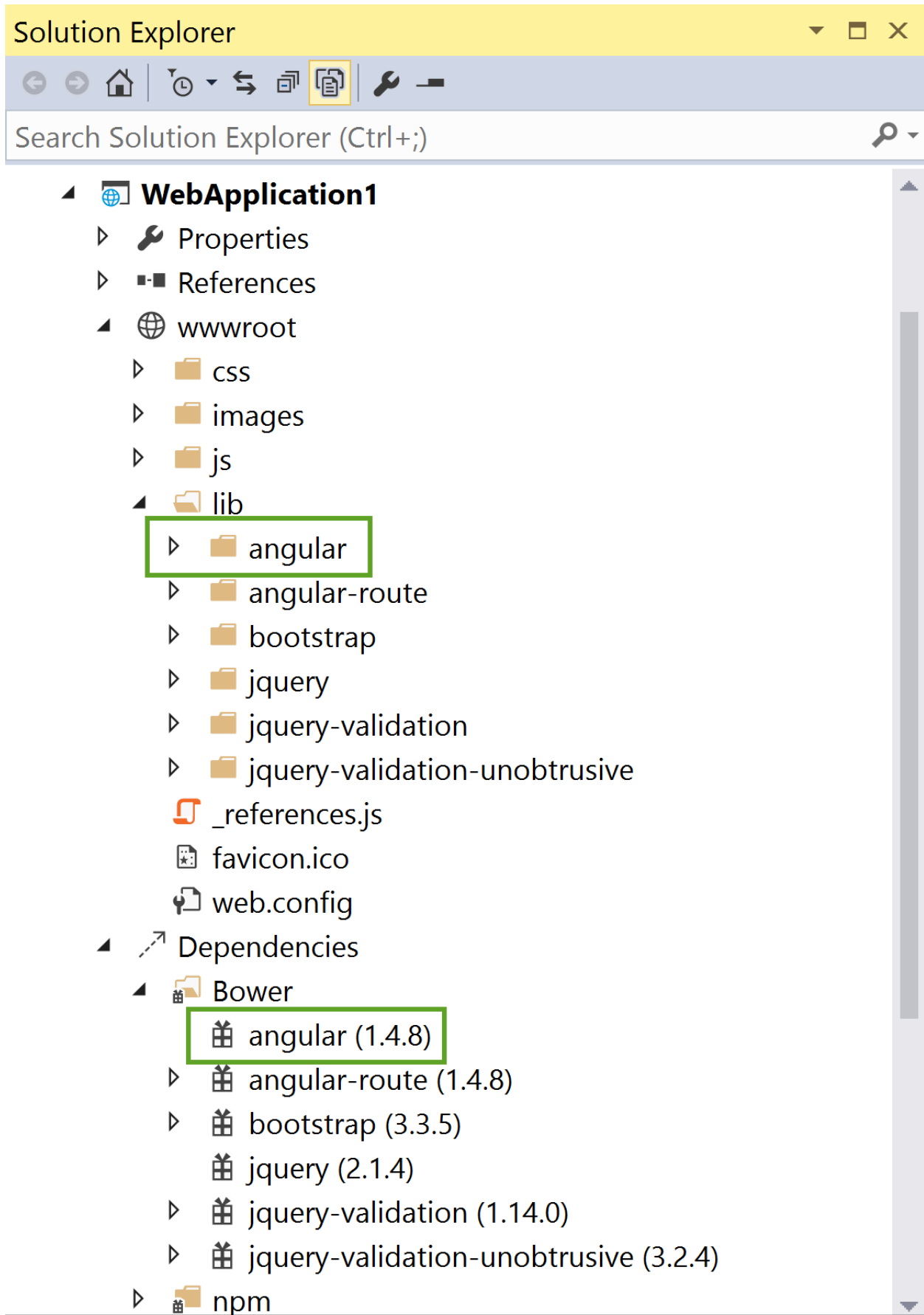
Next, add a `<script>` reference to the bottom of the `<body>` section of your HTML page or *_Layout.cshtml* file, as shown here:

```
1       <environment names="Development">
2           <script src="~/lib/jquery/dist/jquery.js"></script>
3           <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
4           <script src="~/lib/angular/angular.js"></script>
5       </environment>
```

It's recommended that production applications utilize CDNs for common libraries like Angular. You can reference Angular from one of several CDNs, such as this one:

```
1       <environment names="Staging,Production">
2           <script src="//ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"
3                   asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
4                   asp-fallback-test="window.jQuery">
5           </script>
6           <script src="//ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/bootstrap.min.js"
7                   asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
8                   asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
9           </script>
10          <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"
11                  asp-fallback-src="~/lib/angular/angular.min.js"
12                  asp-fallback-test="window.angular">
13          </script>
14          <script src="~/js/site.min.js" asp-append-version="true"></script>
15      </environment>
```

Once you have a reference to the angular.js script file, you're ready to begin using Angular in your web pages.

Key Components

AngularJS includes a number of major components, such as *directives*, *templates*, *repeaters*, *modules*, *controllers*, and more. Let's examine how these components work together to add behavior to your web pages.

Directives

AngularJS uses directives to extend HTML with custom attributes and elements. AngularJS directives are defined via `data-ng-*` or `ng-*` prefixes (`ng` is short for angular). There are two types of AngularJS directives:

1. **Primitive Directives**: These are predefined by the Angular team and are part of the AngularJS framework.

2. **Custom Directives**: These are custom directives that you can define.

One of the primitive directives used in all AngularJS applications is the `ng-app` directive, which bootstraps the AngularJS application. This directive can be applied to the `<body>` tag or to a child element of the body. Let's see an example in action. Assuming you're in an ASP.NET project, you can either add an HTML file to the `wwwroot` folder, or add a new controller action and an associated view. In this case, I've added a new `Directives` action method to `HomeController.cs`. The associated view is shown here:

```
1   @{
2       Layout = "";
3   }
4   <html>
5   <body ng-app>
6       <h1>Directives</h1>
```

```
7       {{2+2}}
8       <script src="~/lib/angular/angular.js"></script>
9   </body>
10  </html>
```

To keep these samples independent of one another, I'm not using the shared layout file. You can see that we decorated the body tag with the `ng-app` directive to indicate this page is an AngularJS application. The `{{2+2}}` is an Angular data binding expression that you will learn more about in a moment. Here is the result if you run this application:
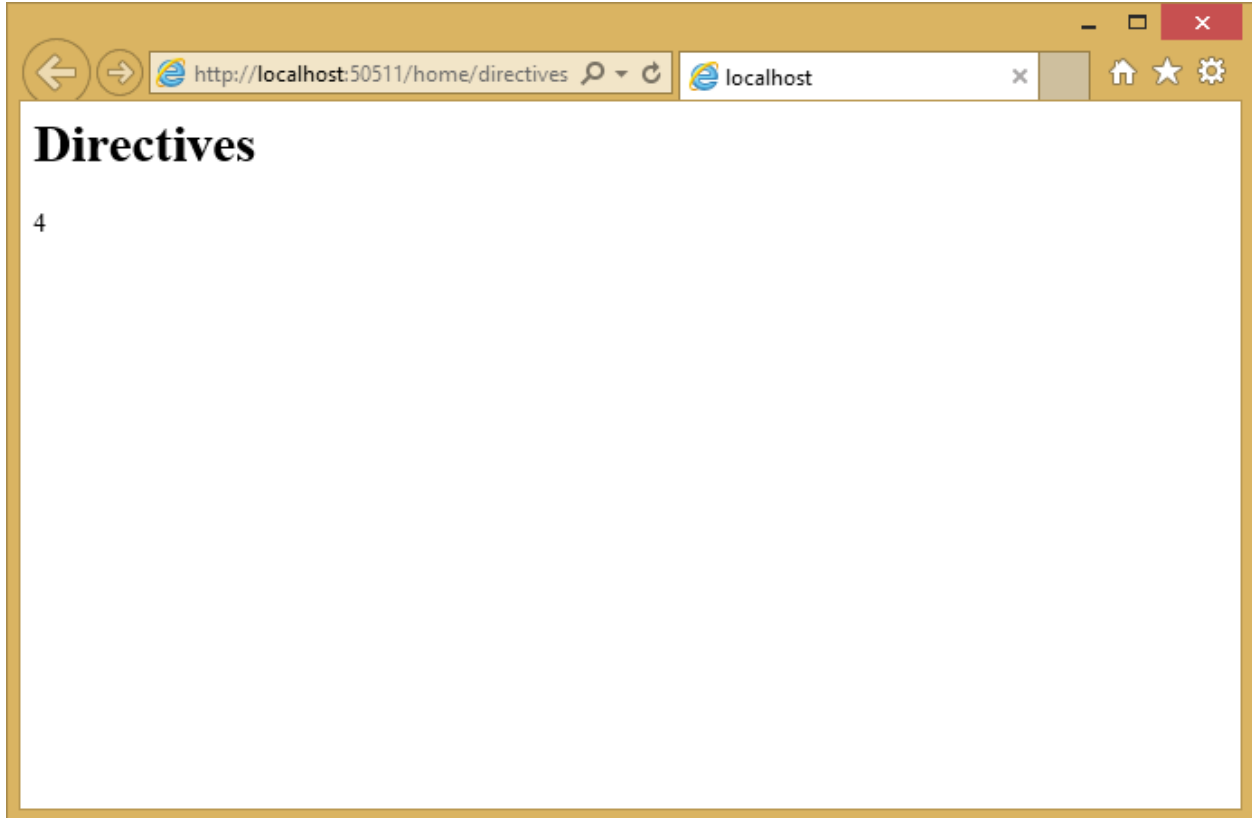
Other primitive directives in AngularJS include:

ng-controller Determines which JavaScript controller is bound to which view.

ng-model Determines the model to which the values of an HTML element's properties are bound.

ng-init Used to initialize the application data in the form of an expression for the current scope.

ng-if Removes or recreates the given HTML element in the DOM based on the truthiness of the expression provided.

ng-repeat Repeats a given block of HTML over a set of data.

ng-show Shows or hides the given HTML element based on the expression provided.

For a full list of all primitive directives supported in AngularJS, please refer to the directive documentation section on the AngularJS documentation website.

Data Binding

AngularJS provides data binding support out-of-the-box using either the `ng-bind` directive or a data binding expression syntax such as `{{expression}}`. AngularJS supports two-way data binding where data from a model is kept

---

in synchronization with a view template at all times. Any changes to the view are automatically reflected in the model. Likewise, any changes in the model are reflected in the view.

Create either an HTML file or a controller action with an accompanying view named `Databinding`. Include the following in the view:

```
@{
    Layout = "";
}
<html>
<body ng-app>
    <h1>Databinding</h1>

    <div ng-init="firstName='John'; lastName='Doe';">
        <strong>First name:</strong> {{firstName}} <br />
        <strong>Last name:</strong> <span ng-bind="lastName" />
    </div>

    <script src="~/lib/angular/angular.js"></script>
</body>
</html>
```

Notice that you can display model values using either directives or data binding (`ng-bind`). The resulting page should look like this:
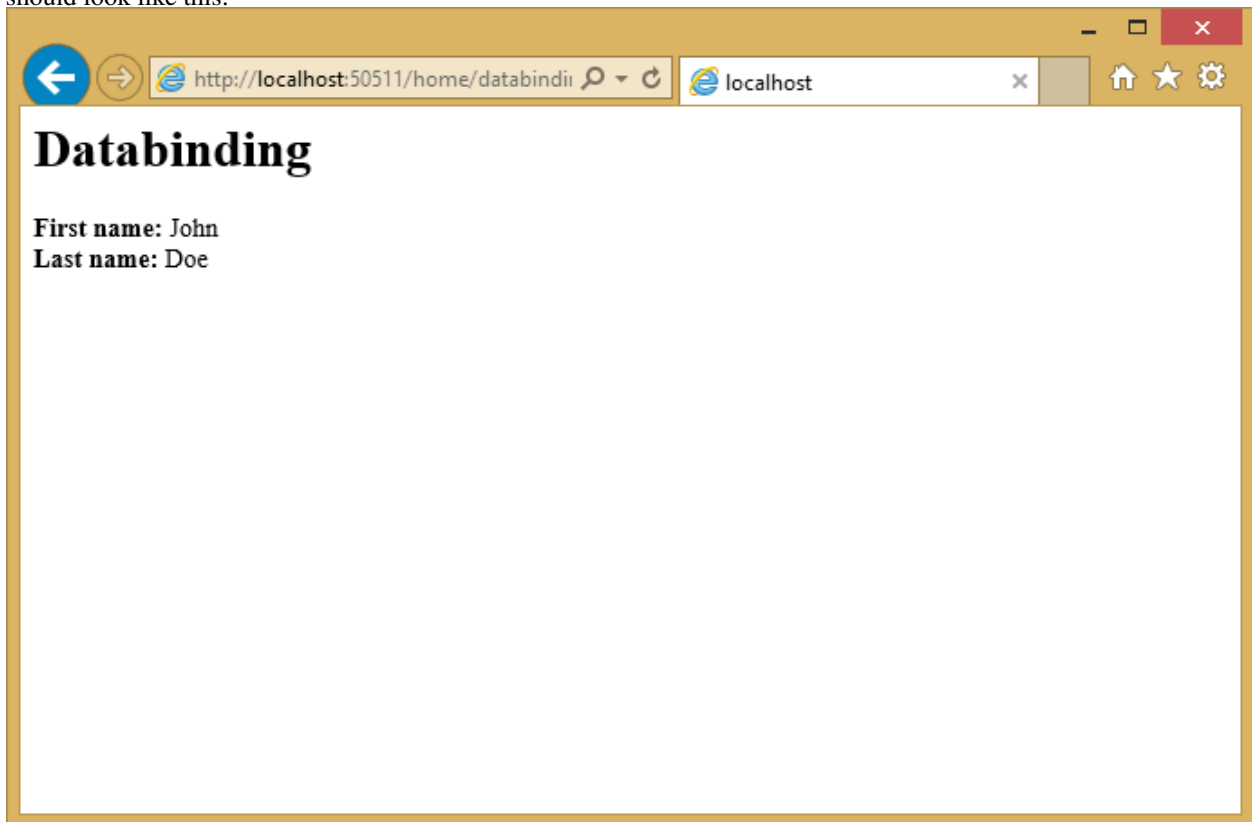
Templates

Templates in AngularJS are just plain HTML pages decorated with AngularJS directives and artifacts. A template in AngularJS is a mixture of directives, expressions, filters, and controls that combine with HTML to form the view.

Add another view to demonstrate templates, and add the following to it:

```
@{
    Layout = "";
}
<html>
<body ng-app>
    <h1>Templates</h1>

    <div ng-init="personName='John Doe'">
        <input ng-model="personName" /> {{personName}}
    </div>

    <script src="~/lib/angular/angular.js"></script>
</body>
</html>
```

The template has AngularJS directives like `ng-app`, `ng-init`, `ng-model` and data binding expression syntax to bind the `personName` property. Running in the browser, the view looks like the screenshot below:
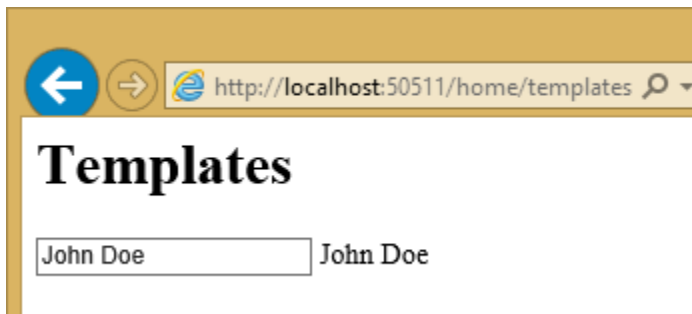
If you change the name by typing in the input field, you will see the text next to the input field dynamically update, showing Angular two-way data binding in action.
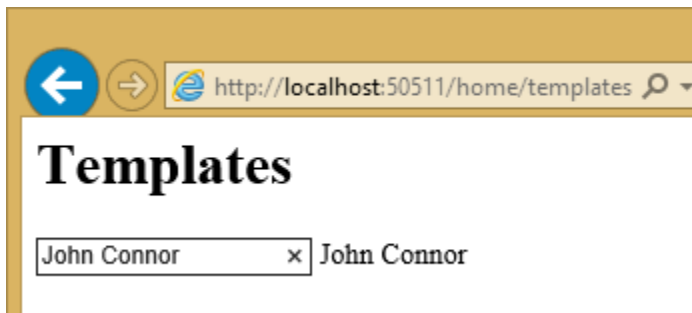
Expressions

Expressions in AngularJS are JavaScript-like code snippets that are written inside the `{{ expression }}` syntax. The data from these expressions is bound to HTML the same way as `ng-bind` directives. The main difference between AngularJS expressions and regular JavaScript expressions is that AngularJS expressions are evaluated against the `$scope` object in AngularJS.

The AngularJS expressions in the sample below bind `personName` and a simple JavaScript calculated expression:

```
@{
    Layout = "";
```

```
3    }
4    <html>
5    <body ng-app>
6        <h1>Expressions</h1>
7
8        <div ng-init="personName='John Doe'">
9            Person's name is: {{personName}} <br />
10           Simple JavaScript calculation of 1 + 2: {{1+2}}
11       </div>
12
13       <script src="~/lib/angular/angular.js"></script>
14   </body>
15   </html>
```

The example running in the browser displays the `personName` data and the results of the calculation:
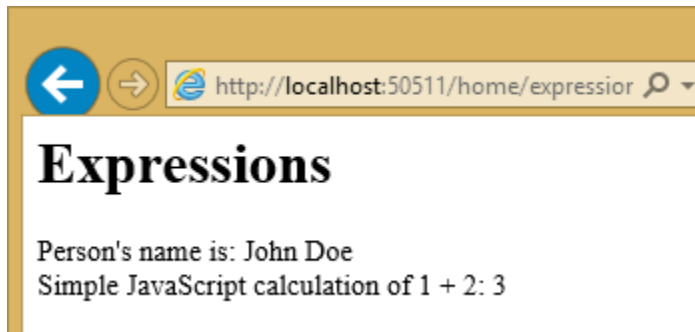
Repeaters

Repeating in AngularJS is done via a primitive directive called `ng-repeat`. The `ng-repeat` directive repeats a given HTML element in a view over the length of a repeating data array. Repeaters in AngularJS can repeat over an array of strings or objects. Here is a sample usage of repeating over an array of strings:

```
1    @{
2        Layout = "";
3    }
4    <html>
5    <body ng-app>
6        <h1>Repeaters</h1>
7
8        <div ng-init="names=['John Doe', 'Mary Jane', 'Bob Parker']">
9            <ul>
10               <li ng-repeat="name in names">
11                   {{name}}
12               </li>
13           </ul>
14       </div>
15
16       <script src="~/lib/angular/angular.js"></script>
17   </body>
18   </html>
```

The repeat directive outputs a series of list items in an unordered list, as you can see in the developer tools shown in this screenshot:
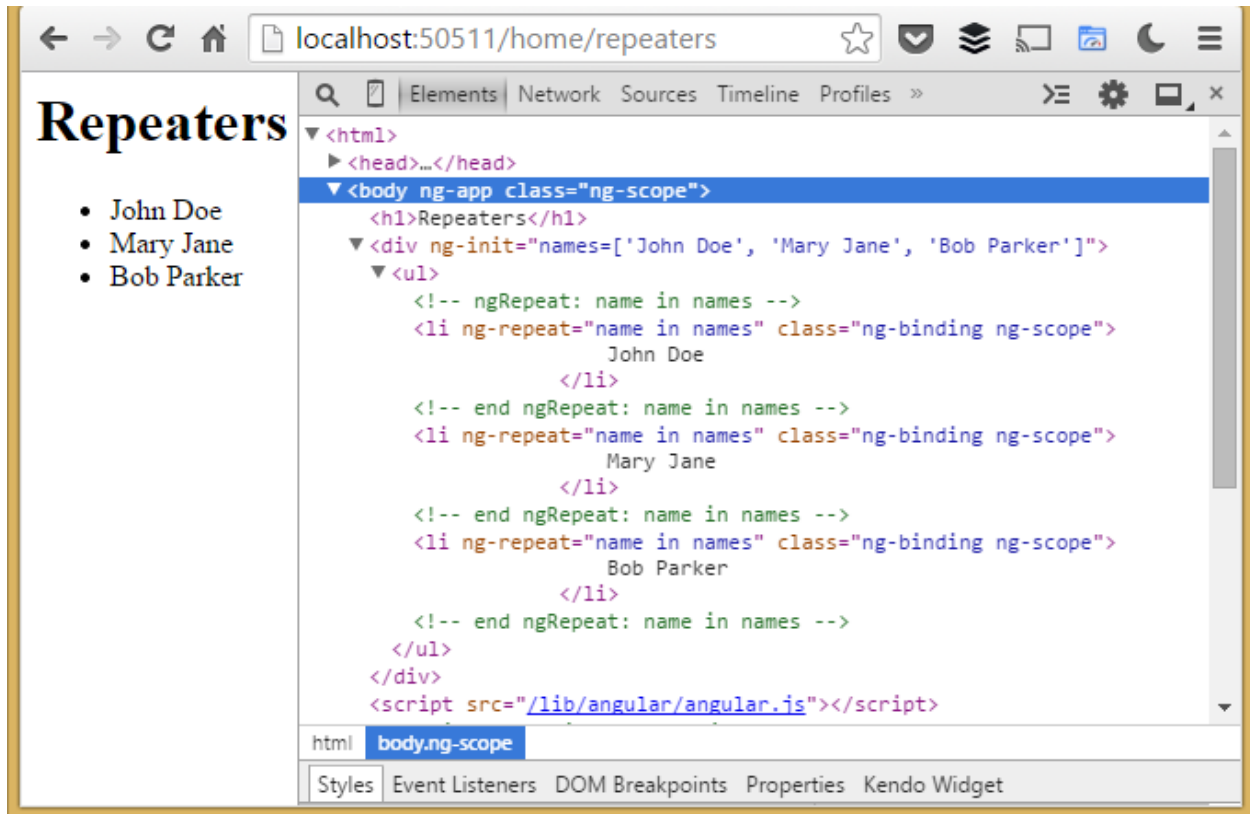
Here is an example that repeats over an array of objects. The `ng-init` directive establishes a `names` array, where each element is an object containing first and last names. The `ng-repeat` assignment, `name in names`, outputs a list item for every array element.

```
1  @{
2      Layout = "";
3  }
4  <html>
5  <body ng-app>
6      <h1>Repeaters2</h1>
7
8      <div ng-init="names=[
9                  {firstName:'John', lastName:'Doe'},
10                 {firstName:'Mary', lastName:'Jane'},
11                 {firstName:'Bob', lastName:'Parker'}]">
12         <ul>
13             <li ng-repeat="name in names">
14                 {{name.firstName + ' ' + name.lastName}}
15             </li>
16         </ul>
17     </div>
18
19     <script src="~/lib/angular/angular.js"></script>
20 </body>
21 </html>
```

The output in this case is the same as in the previous example.

Angular provides some additional directives that can help provide behavior based on where the loop is in its execution.

$index Use `$index` in the `ng-repeat` loop to determine which index position your loop currently is on.

$even and $odd Use $even in the ng-repeat loop to determine whether the current index in your loop is an even indexed row. Similarly, use $odd to determine if the current index is an odd indexed row.

$first and $last Use $first in the ng-repeat loop to determine whether the current index in your loop is the first row. Similarly, use $last to determine if the current index is the last row.

Below is a sample that shows $index, $even, $odd, $first, and $last in action:

```
@{
    Layout = "";
}
<html>
<body ng-app>
    <h1>Repeaters2</h1>

    <div ng-init="names=[
                {firstName:'John', lastName:'Doe'},
                {firstName:'Mary', lastName:'Jane'},
                {firstName:'Bob', lastName:'Parker'}]">
        <ul>
            <li ng-repeat="name in names">
                {{name.firstName + ' ' + name.lastName}} at index {{$index}}
                <span ng-show="{{$first}}">, the first position</span>
                <span ng-show="{{$last}}">, the last position</span>
                <span ng-show="{{$odd}}">,which is odd-numbered.</span>
                <span ng-show="{{$even}}">,which is even-numbered.</span>
            </li>
        </ul>
    </div>

    <script src="~/lib/angular/angular.js"></script>
</body>
</html>
```

Here is the resulting output:
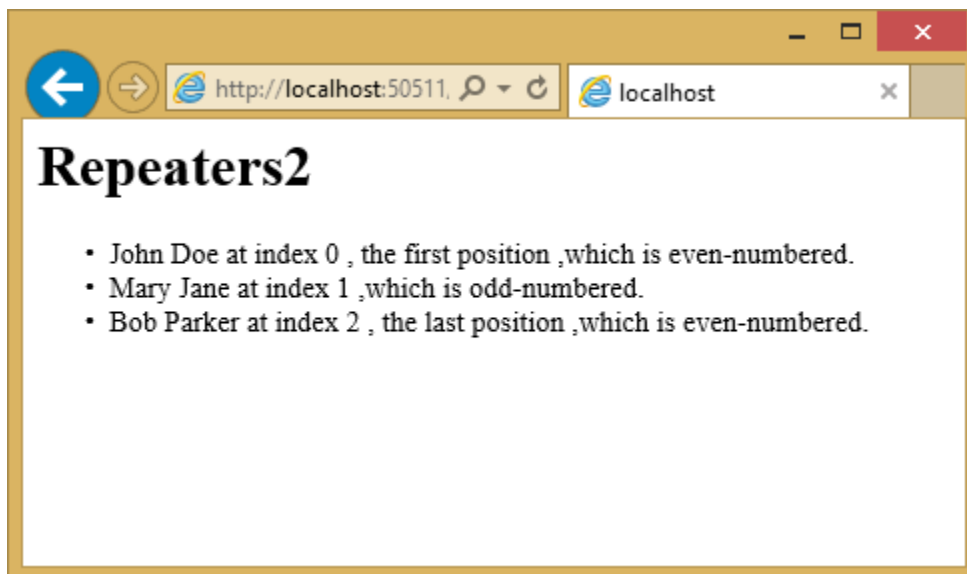
$scope

$scope is a JavaScript object that acts as glue between the view (template) and the controller (explained below). A view template in AngularJS only knows about the values attached to the $scope object in the controller.

: In the MVVM world, the $scope object in AngularJS is often defined as the ViewModel. The AngularJS team refers to the $scope object as the Data-Model. Learn more about Scopes in AngularJS.

Below is a simple example showing how to set properties on $scope within a separate JavaScript file, scope.js:

```
1  var personApp = angular.module('personApp', []);
2  personApp.controller('personController', ['$scope', function ($scope) {
3      $scope.name = 'Mary Jane';
4  }]);
```

Observe the $scope parameter passed to the controller on line 2. This object is what the view knows about. On line 3, we are setting a property called "name" to "Mary Jane".

What happens when a particular property is not found by the view? The view defined below refers to "name" and "age" properties:

```
1  @{
2      Layout = "";
3  }
4  <html>
5  <body ng-app="personApp">
6      <h1>Scope</h1>
7
8      <div ng-controller="personController">
9          <strong>Name:</strong> {{name}} <br />
10         <strong>Missing Property (age):</strong> {{age}}
11     </div>
12
13     <script src="~/lib/angular/angular.js"></script>
14     <script src="~/app/scope.js"></script>
15 </body>
16 </html>
```

Notice on line 9 that we are asking Angular to show the "name" property using expression syntax. Line 10 then refers to "age", a property that does not exist. The running example shows the name set to "Mary Jane" and nothing for age. Missing properties are ignored.
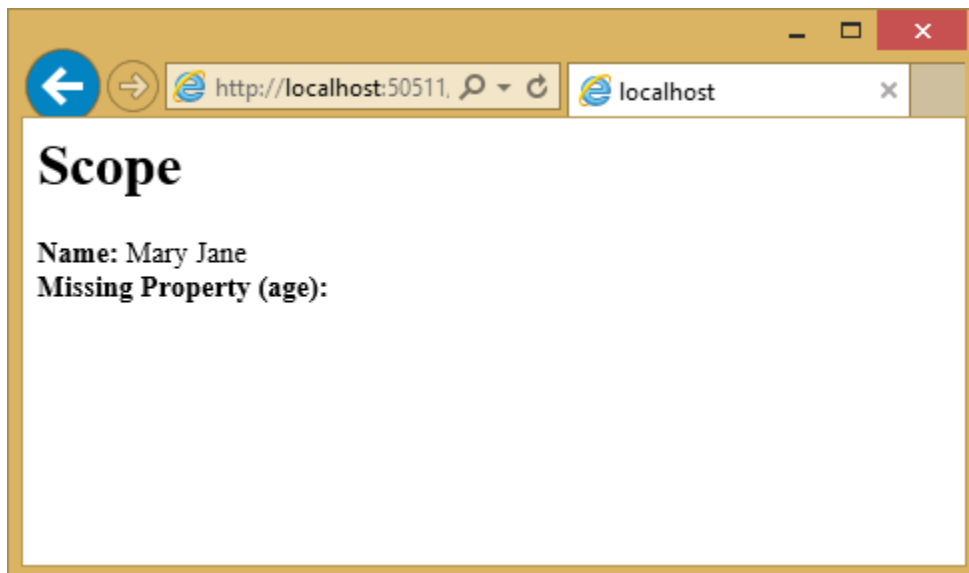
Modules

A module in AngularJS is a collection of controllers, services, directives, etc. The `angular.module()` function call is used to create, register, and retrieve modules in AngularJS. All modules, including those shipped by the AngularJS team and third party libraries, should be registered using the `angular.module()` function.

Below is a snippet of code that shows how to create a new module in AngularJS. The first parameter is the name of the module. The second parameter defines dependencies on other modules. Later in this article, we will be showing how to pass these dependencies to an `angular.module()` method call.

```
var personApp = angular.module('personApp', []);
```

Use the `ng-app` directive to represent an AngularJS module on the page. To use a module, assign the name of the module, `personApp` in this example, to the `ng-app` directive in our template.

```
<body ng-app="personApp">
```

Controllers

Controllers in AngularJS are the first point of entry for your code. The `<module name>.controller()` function call is used to create and register controllers in AngularJS. The `ng-controller` directive is used to represent an AngularJS controller on the HTML page. The role of the controller in Angular is to set state and behavior of the data model (`$scope`). Controllers should not be used to manipulate the DOM directly.

Below is a snippet of code that registers a new controller. The `personApp` variable in the snippet references an Angular module, which is defined on line 2.

```
1  // module
2  var personApp = angular.module('personApp', []);
3
4  // controller
5  personApp.controller('personController', function ($scope) {
6      $scope.firstName = "Mary";
```

```
7        $scope.lastName = "Jane"
8    });
```

The view using the `ng-controller` directive assigns the controller name:

```
1    @{
2        Layout = "";
3    }
4    <html>
5    <body ng-app="personApp">
6        <h1>Controllers</h1>
7
8        <div ng-controller="personController">
9            <strong>First Name:</strong> {{firstName}} <br />
10           <strong>Last Name:</strong> {{lastName}}
11       </div>
12
13       <script src="~/lib/angular/angular.js"></script>
14       <script src="~/app/controllers.js"></script>
15   </body>
16   </html>
```

The page shows "Mary" and "Jane" that correspond to the `firstName` and `lastName` properties attached to the `$scope` object:
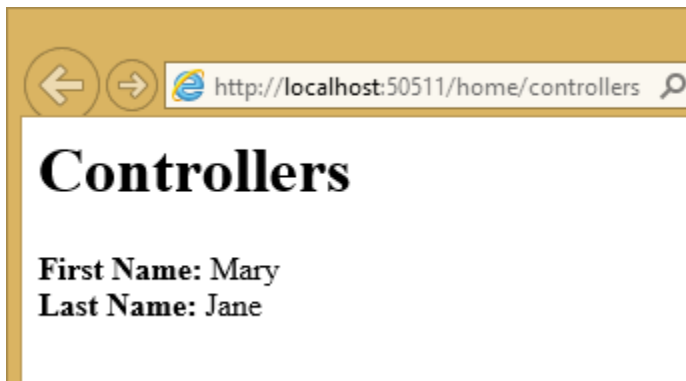
Services

Services in AngularJS are commonly used for shared code that is abstracted away into a file which can be used throughout the lifetime of an Angular application. Services are lazily instantiated, meaning that there will not be an instance of a service unless a component that depends on the service gets used. Factories are an example of a service used in AngularJS applications. Factories are created using the `myApp.factory()` function call, where `myApp` is the module.

Below is an example that shows how to use factories in AngularJS:

```
1    personApp.factory('personFactory', function () {
2        function getName() {
3            return "Mary Jane";
4        }
5
6        var service = {
7            getName: getName
```

```
8          };
9
10         return service;
11   });
```

To call this factory from the controller, pass `personFactory` as a parameter to the `controller` function:

```
personApp.controller('personController', function($scope,personFactory) {
    $scope.name = personFactory.getName();
});
```

Using services to talk to a REST endpoint

Below is an end-to-end example using services in AngularJS to interact with an ASP.NET Core Web API endpoint. The example gets data from the Web API and displays the data in a view template. Let's start with the view first:

```
1    @{
2        Layout = "";
3    }
4    <html>
5    <body ng-app="PersonsApp">
6        <h1>People</h1>
7
8        <div ng-controller="personController">
9            <ul>
10               <li ng-repeat="person in people">
11                   <h2>{{person.FirstName}} {{person.LastName}}</h2>
12               </li>
13           </ul>
14       </div>
15
16       <script src="~/lib/angular/angular.js"></script>
17       <script src="~/app/personApp.js"></script>
18       <script src="~/app/personFactory.js"></script>
19       <script src="~/app/personController.js"></script>
20   </body>
21   </html>
```

In this view, we have an Angular module called `PersonsApp` and a controller called `personController`. We are using `ng-repeat` to iterate over the list of persons. We are referencing three custom JavaScript files on lines 17-19.

The `personApp.js` file is used to register the `PersonsApp` module; and, the syntax is similar to previous examples. We are using the `angular.module` function to create a new instance of the module that we will be working with.

```
1    (function () {
2        'use strict';
3        var app = angular.module('PersonsApp', []);
4    })();
```

Let's take a look at `personFactory.js`, below. We are calling the module's `factory` method to create a factory. Line 12 shows the built-in Angular `$http` service retrieving people information from a web service.

```
1  (function () {
2      'use strict';
3
4      var serviceId = 'personFactory';
5
6      angular.module('PersonsApp').factory(serviceId,
7          ['$http', personFactory]);
8
9      function personFactory($http) {
10
11          function getPeople() {
12              return $http.get('/api/people');
13          }
14
15          var service = {
16              getPeople: getPeople
17          };
18
19          return service;
20      }
21  })();
```

In `personController.js`, we are calling the module's `controller` method to create the controller. The `$scope` object's `people` property is assigned the data returned from the personFactory (line 13).

```
1  (function () {
2      'use strict';
3
4      var controllerId = 'personController';
5
6      angular.module('PersonsApp').controller(controllerId,
7          ['$scope', 'personFactory', personController]);
8
9      function personController($scope, personFactory) {
10          $scope.people = [];
11
12          personFactory.getPeople().success(function (data) {
13              $scope.people = data;
14          }).error(function (error) {
15              // log errors
16          });
17      }
18  })();
```

Let's take a quick look at the Web API and the model behind it. The `Person` model is a POCO (Plain Old CLR Object) with `Id`, `FirstName`, and `LastName` properties:

```
1  namespace AngularSample.Models
2  {
3      public class Person
4      {
5          public int Id { get; set; }
6          public string FirstName { get; set; }
7          public string LastName { get; set; }
8      }
9  }
```

The `Person` controller returns a JSON-formatted list of `Person` objects:

```
1  using AngularSample.Models;
2  using Microsoft.AspNet.Mvc;
3  using System.Collections.Generic;
4
5  namespace AngularSample.Controllers.Api
6  {
7      public class PersonController : Controller
8      {
9          [Route("/api/people")]
10         public JsonResult GetPeople()
11         {
12             var people = new List<Person>()
13             {
14                 new Person { Id = 1, FirstName = "John", LastName = "Doe" },
15                 new Person { Id = 1, FirstName = "Mary", LastName = "Jane" },
16                 new Person { Id = 1, FirstName = "Bob", LastName = "Parker" }
17             };
18
19             return Json(people);
20         }
21     }
22 }
```

Let's see the application in action:
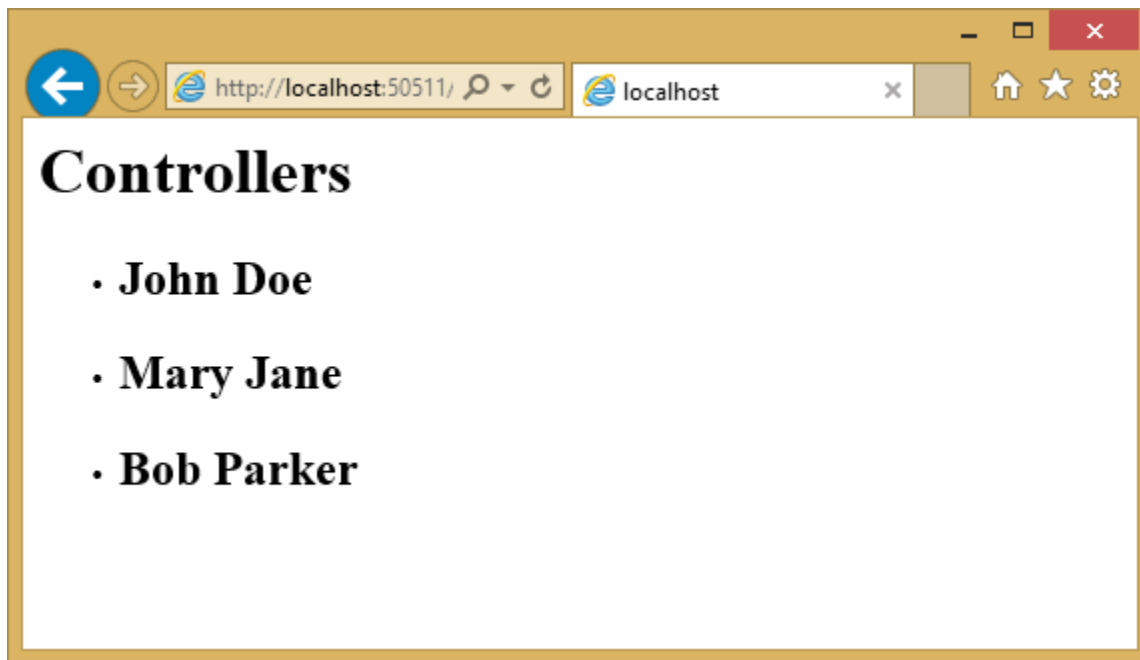
You can view the application's structure on GitHub.

: For more on structuring AngularJS applications, see John Papa's Angular Style Guide

---

: To create AngularJS module, controller, factory, directive and view files easily, be sure to check out Sayed Hashimi's SideWaffle template pack for Visual Studio. Sayed Hashimi is a Senior Program Manager on the Visual Studio Web Team at Microsoft and SideWaffle templates are considered the gold standard. At the time of this writing, SideWaffle

---

is available for Visual Studio 2012, 2013, and 2015.

Routing and Multiple Views

AngularJS has a built-in route provider to handle SPA (Single Page Application) based navigation. To work with routing in AngularJS, you must add the `angular-route` library using Bower. You can see in the *bower.json* file referenced at the start of this article that we are already referencing it in our project.

After you install the package, add the script reference (`angular-route.js`) to your view.

Now let's take the Person App we have been building and add navigation to it. First, we will make a copy of the app by creating a new `PeopleController` action called `Spa` and a corresponding `Spa.cshtml` view by copying the `Index.cshtml` view in the `People` folder. Add a script reference to `angular-route` (see line 11). Also add a `div` marked with the `ng-view` directive (see line 6) as a placeholder to place views in. We are going to be using several additional `.js` files which are referenced on lines 13-16.

```
1  @{
2      Layout = "";
3  }
4  <html>
5  <body ng-app="personApp">
6      <div ng-view>
7
8      </div>
9
10     <script src="~/lib/angular/angular.js"></script>
11     <script src="~/lib/angular-route/angular-route.js"></script>
12
13     <script src="~/app/personModule.js"></script>
14     <script src="~/app/personRoutes.js"></script>
15     <script src="~/app/personListController.js"></script>
16     <script src="~/app/personDetailController.js"></script>
17 </body>
18 </html>
```

Let's take a look at `personModule.js` file to see how we are instantiating the module with routing. We are passing `ngRoute` as a library into the module. This module handles routing in our application.

```
1  var personApp = angular.module('personApp', ['ngRoute']);
```

The `personRoutes.js` file, below, defines routes based on the route provider. Lines 4-7 define navigation by effectively saying, when a URL with `/persons` is requested, use a template called `partials/personlist` by working through `personListController`. Lines 8-11 indicate a detail page with a route parameter of `personId`. If the URL doesn't match one of the patterns, Angular defaults to the `/persons` view.

```
1  personApp.config(['$routeProvider',
2      function ($routeProvider) {
3          $routeProvider.
4              when('/persons', {
5                  templateUrl: '/app/partials/personlist.html',
6                  controller: 'personListController'
7              }).
8              when('/persons/:personId', {
9                  templateUrl: '/app/partials/persondetail.html',
10                 controller: 'personDetailController'
```

```
11          }).
12          otherwise({
13              redirectTo: '/persons'
14          })
15      }
16  ]);
```

The `personlist.html` file is a partial view containing only the HTML needed to display person list.

```
1  <div>
2      <h1>PERSONS PAGE</h1>
3      <span ng-bind="message"/>
4  </div>
```

The controller is defined by using the module's `controller` function in `personListController.js`.

```
1  personApp.controller('personListController', function ($scope) {
2      $scope.message = "You are on the Persons List Page.";
3  })
```

If we run this application and navigate to the `people/spa#/persons` URL, we will see:
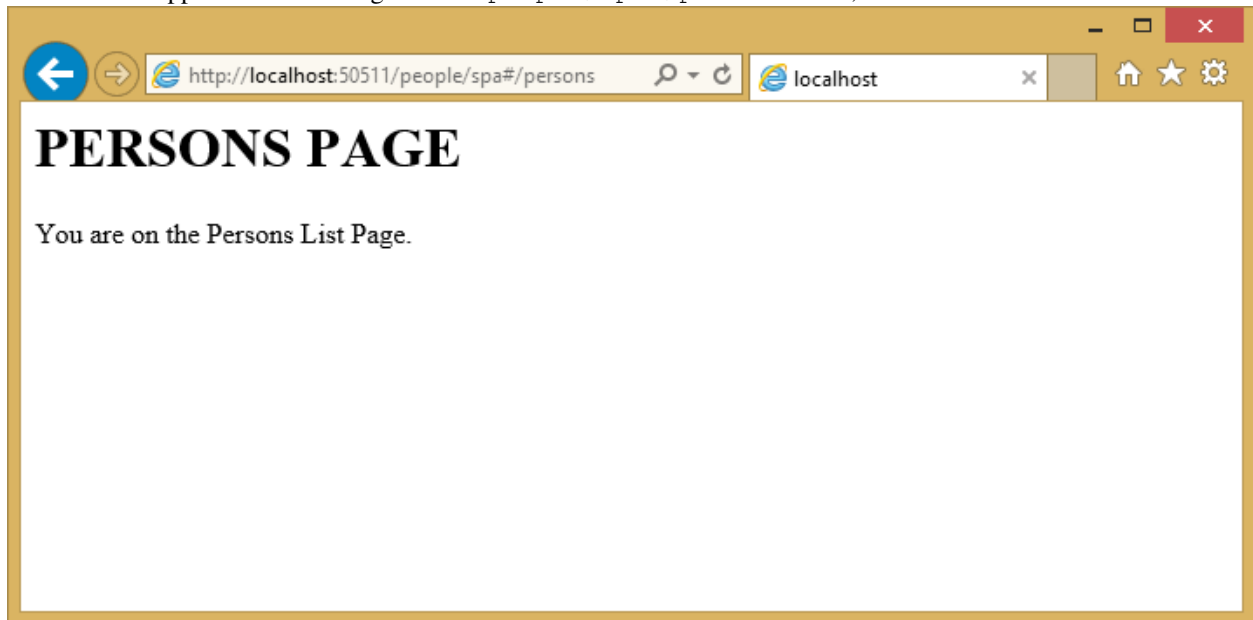
If we navigate to a detail page, for example `people/spa#/persons/2`, we will see the detail partial view:
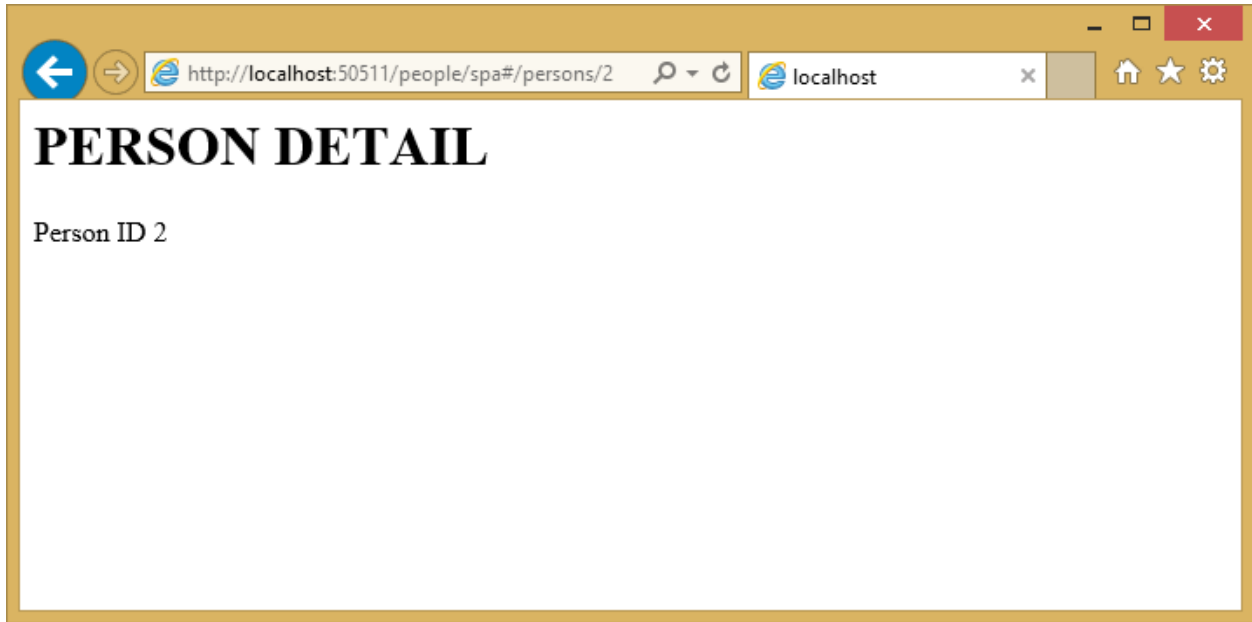
You can view the full source and any files not shown in this article on GitHub.

Event Handlers

There are a number of directives in AngularJS that add event-handling capabilities to the input elements in your HTML DOM. Below is a list of the events that are built into AngularJS.

- ng-click
- ng-dbl-click
- ng-mousedown
- ng-mouseup
- ng-mouseenter
- ng-mouseleave
- ng-mousemove
- ng-keydown
- ng-keyup
- ng-keypress
- ng-change

: You can add your own event handlers using the custom directives feature in AngularJS.

Let's look at how the ng-click event is wired up. Create a new JavaScript file named eventHandlerController.js, and add the following to it:

```
personApp.controller('eventHandlerController', function ($scope) {
    $scope.firstName = 'Mary';
    $scope.lastName = 'Jane';
```

```
4
5          $scope.sayName = function () {
6              alert('Welcome, ' + $scope.firstName + ' ' + $scope.lastName);
7          }
8      });
```

Notice the new `sayName` function in `eventHandlerController` on line 5 above. All the method is doing for now is showing a JavaScript alert to the user with a welcome message.

The view below binds a controller function to an AngularJS event. Line 9 has a button on which the `ng-click` Angular directive has been applied. It calls our `sayName` function, which is attached to the `$scope` object passed to this view.

```
1   @{
2       Layout = "";
3   }
4   <html>
5   <body ng-app="personApp">
6       <div ng-controller="eventHandlerController">
7           <strong>First Name:</strong> {{firstName}} <br />
8           <strong>Last Name:</strong> {{lastName}} <br />
9           <input ng-click="sayName()" type="button" value="Say Name" />
10      </div>
11      <script src="~/lib/angular/angular.js"></script>
12      <script src="~/lib/angular-route/angular-route.js"></script>
13
14      <script src="~/app/personModule.js"></script>
15      <script src="~/app/eventHandlerController.js"></script>
16  </body>
17  </html>
```

The running example demonstrates that the controller's `sayName` function is called automatically when the button is clicked.
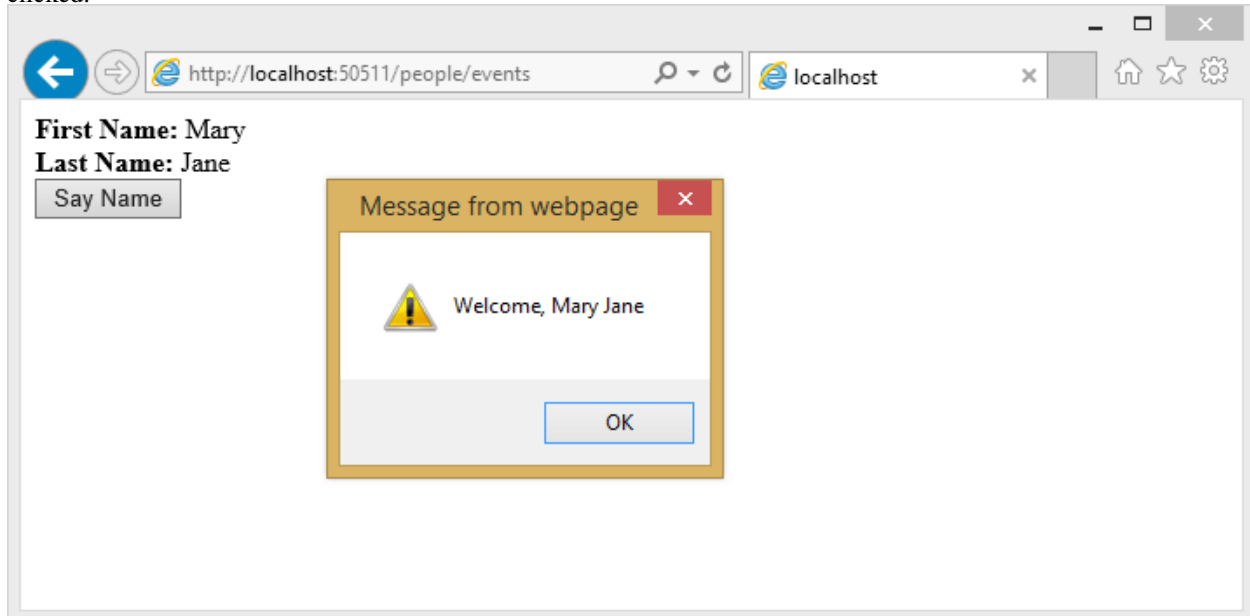
For more detail on AngularJS built-in event handler directives, be sure to head to the documentation website of AngularJS.

Angular 2.0

Angular 2.0 is the next version of AngularJS, which is completely reimagined with ES6 and mobile in mind. It's built using Microsoft's TypeScript language. Angular 2.0 is currently a RC product and is expected to be released in early 2016. Several breaking changes will be introduced in the Angular 2.0 release, so the Angular team is working hard to provide guidance to developers. A migration path will become more clear as the release date approaches. If you wish to play with Angular 2.0 now, the Angular team has created Angular.io to show their progress, to provide early documentation, and to gather feedback.

Summary

This article provides an overview of AngularJS for ASP.NET developers. It aims to help developers who are new to this SPA framework get up-to-speed quickly.

Related Resources

- Angular Docs
- Angular 2 Info

1.8.7 Styling Applications with Less, Sass, and Font Awesome

By Steve Smith

Users of web applications have increasingly high expectations when it comes to style and overall experience. Modern web applications frequently leverage rich tools and frameworks for defining and managing their look and feel in a consistent manner. Frameworks like Bootstrap can go a long way toward defining a common set of styles and layout options for the web sites. However, most non-trivial sites also benefit from being able to effectively define and maintain styles and cascading style sheet (CSS) files, as well as having easy access to non-image icons that help make the site's interface more intuitive. That's where languages and tools that support Less and Sass, and libraries like Font Awesome, come in.

Sections:
- *CSS Preprocessor Languages*
- *Less*
- *Sass*
- *Less or Sass?*
- *Font Awesome*
- *Summary*

CSS Preprocessor Languages

Languages that are compiled into other languages, in order to improve the experience of working with the underlying language, are referred to as pre-processors. There are two popular pre-processors for CSS: Less and Sass. These pre-processors add features to CSS, such as support for variables and nested rules, which improve the maintainability of large, complex stylesheets. CSS as a language is very basic, lacking support even for something as simple as variables, and this tends to make CSS files repetitive and bloated. Adding real programming language features via preprocessors can help reduce duplication and provide better organization of styling rules. Visual Studio provides built-in support for both Less and Sass, as well as extensions that can further improve the development experience when working with these languages.

As a quick example of how preprocessors can improve readability and maintainability of style information, consider this CSS:

```
.header {
   color: black;
   font-weight: bold;
   font-size: 18px;
   font-family: Helvetica, Arial, sans-serif;
}

.small-header {
   color: black;
   font-weight: bold;
   font-size: 14px;
   font-family: Helvetica, Arial, sans-serif;
}
```

Using Less, this can be rewritten to eliminate all of the duplication, using a mixin (so named because it allows you to "mix in" properties from one class or rule-set into another):

```
.header {
   color: black;
   font-weight: bold;
   font-size: 18px;
   font-family: Helvetica, Arial, sans-serif;
}

.small-header {
   .header;
   font-size: 14px;
}
```

Visual Studio adds a great deal of built-in support for Less and Sass. You can also add support for earlier versions of Visual Studio by installing the Web Essentials extension.

Less

The Less CSS pre-processor runs using Node.js. You can quickly install it using the Node Package Manager (NPM), with:

```
npm install -g less
```

If you're using Visual Studio, you can get started with Less by adding one or more Less files to your project, and then configuring Gulp (or Grunt) to process them at compile-time. Add a Styles folder to your project, and then add a new Less file called main.less to this folder.
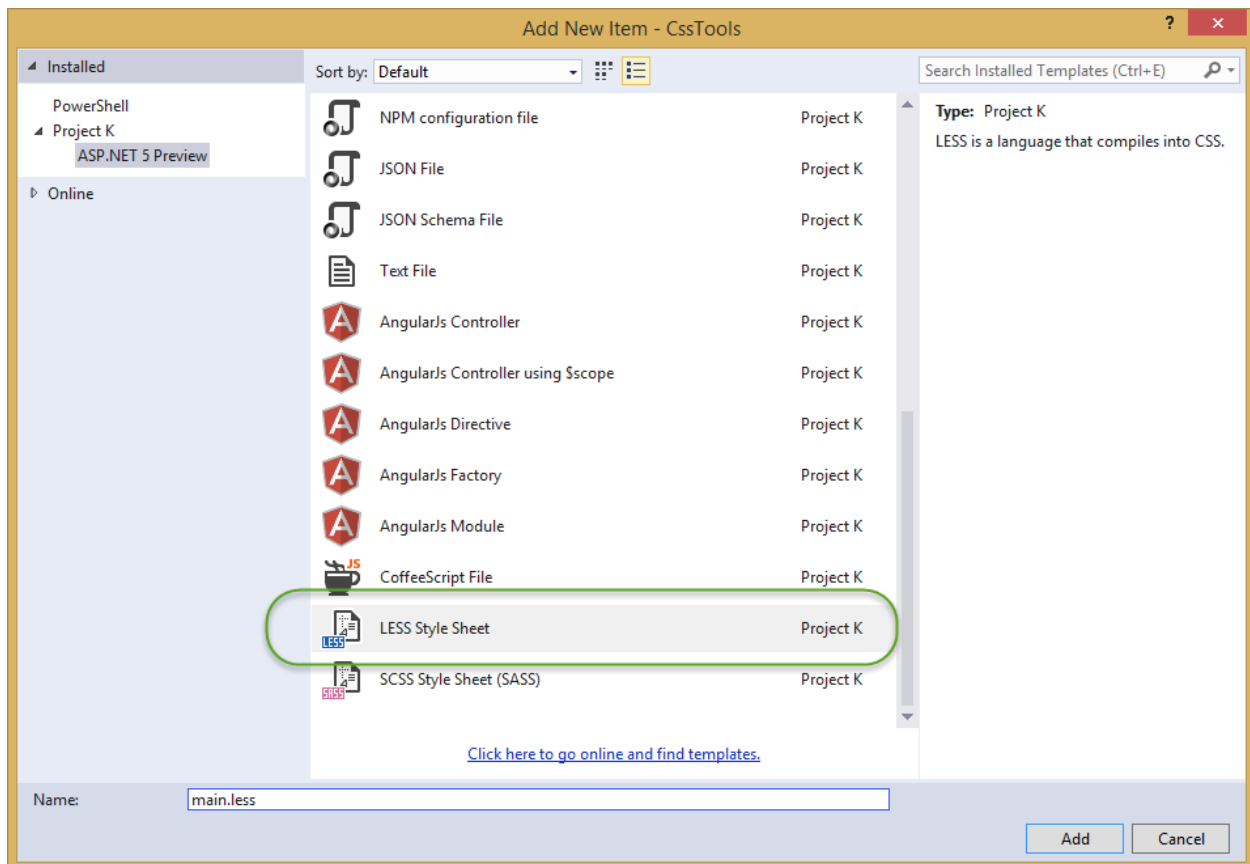
Once added, your folder structure should look something like this:
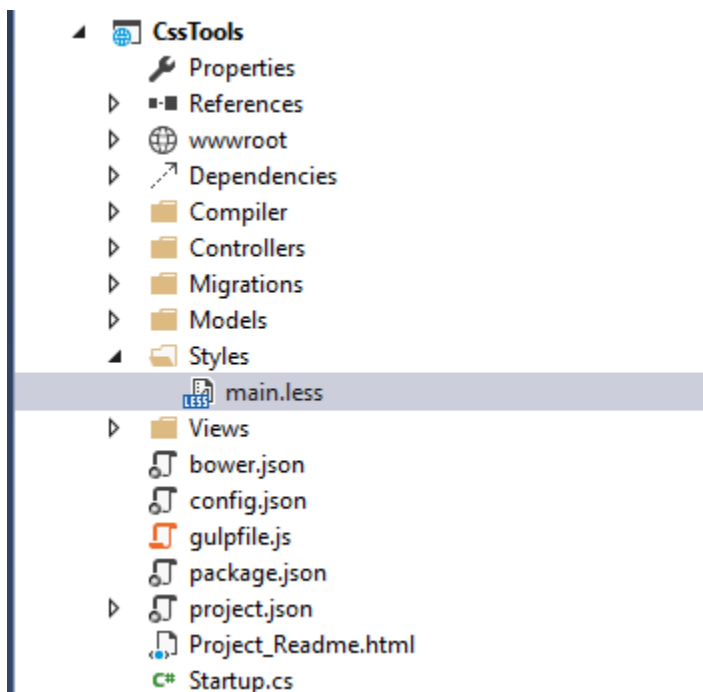
Now we can add some basic styling to the file, which will be compiled into CSS and deployed to the wwwroot folder by Gulp.

Modify main.less to include the following content, which creates a simple color palette from a single base color.

```
@base: #663333;
@background: spin(@base, 180);
@lighter: lighten(spin(@base, 5), 10%);
@lighter2: lighten(spin(@base, 10), 20%);
@darker: darken(spin(@base, -5), 10%);
@darker2: darken(spin(@base, -10), 20%);

body {
  background-color:@background;
}
.baseColor  {color:@base}
.bgLight    {color:@lighter}
.bgLight2   {color:@lighter2}
.bgDark     {color:@darker}
.bgDark2    {color:@darker2}
```

`@base` and the other @-prefixed items are variables. Each of them represents a color. Except for `@base`, they are set using color functions: lighten, darken, and spin. Lighten and darken do pretty much what you would expect; spin adjusts the hue of a color by a number of degrees (around the color wheel). The less processor is smart enough to ignore variables that aren't used, so to demonstrate how these variables work, we need to use them somewhere. The classes `.baseColor`, etc. will demonstrate the calculated values of each of the variables in the CSS file that is produced.

Getting Started

If you don't already have one in your project, add a new Gulp configuration file. Make sure package.json includes gulp in its `devDependencies`, and add "gulp-less":

```
"devDependencies": {
    "gulp": "3.8.11",
    "gulp-less": "3.0.2",
    "rimraf": "2.3.2"
  }
```

Save your changes to the package.json file, and you should see that the all of the files referenced can be found in the Dependencies folder under NPM. If not, right-click on the NPM folder and select "Restore Packages."

Now open gulpfile.js. Add a variable at the top to represent less:

```
var gulp = require("gulp"),
         rimraf = require("rimraf"),
         fs = require("fs"),
         less = require("gulp-less");
```

add another variable to allow you to access project properties:

```
var project = require('./project.json');
```

Next, add a task to run less, using the syntax shown here:

```
gulp.task("less", function () {
  return gulp.src('Styles/main.less')
    .pipe(less())
```

```
    .pipe(gulp.dest(project.webroot + '/css'));
});
```

Open the Task Runner Explorer (view>Other Windows > Task Runner Explorer). Among the tasks, you should see a new task named `less`. Run it, and you should have output similar to what is shown here:
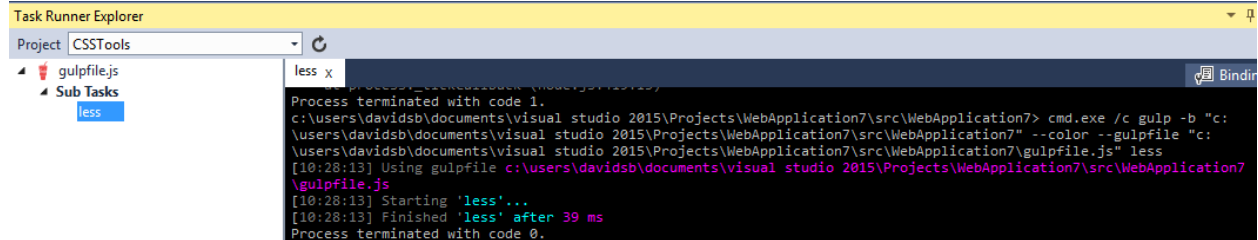
Now refresh your Solution Explorer and inspect the contents of the wwwroot/css folder. You should find a new file, main.css, there:
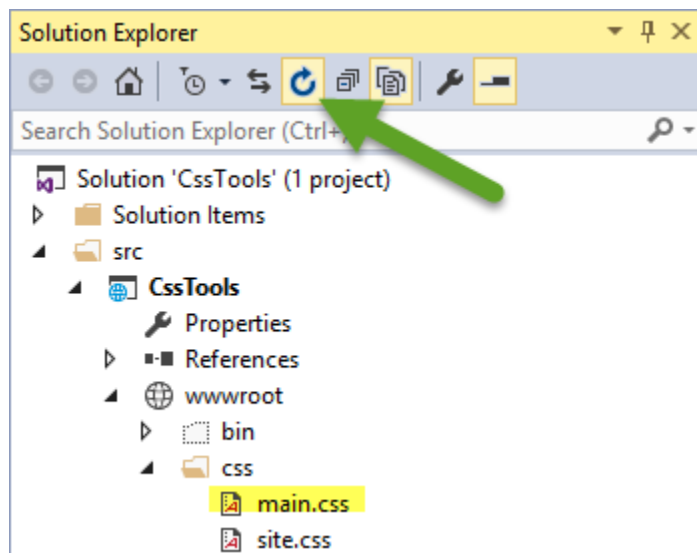
Open main.css and you should see something like the following:

```css
body {
  background-color: #336666;
}
.baseColor {
  color: #663333;
}
.bgLight {
  color: #884a44;
}
.bgLight2 {
  color: #aa6355;
}
.bgDark {
  color: #442225;
}
.bgDark2 {
  color: #221114;
}
```

Add a simple HTML page to the wwwroot folder and reference main.css to see the color palette in action.

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <link href="css/main.css" rel="stylesheet" />
  <title></title>
</head>
<body>
  <div>
    <div class="baseColor">BaseColor</div>
    <div class="bgLight">Light</div>
    <div class="bgLight2">Light2</div>
    <div class="bgDark">Dark</div>
    <div class="bgDark2">Dark2</div>
  </div>
</body>
</html>
```

You can see that the 180 degree spin on `@base` used to produce `@background` resulted in the color wheel opposing color of `@base`:
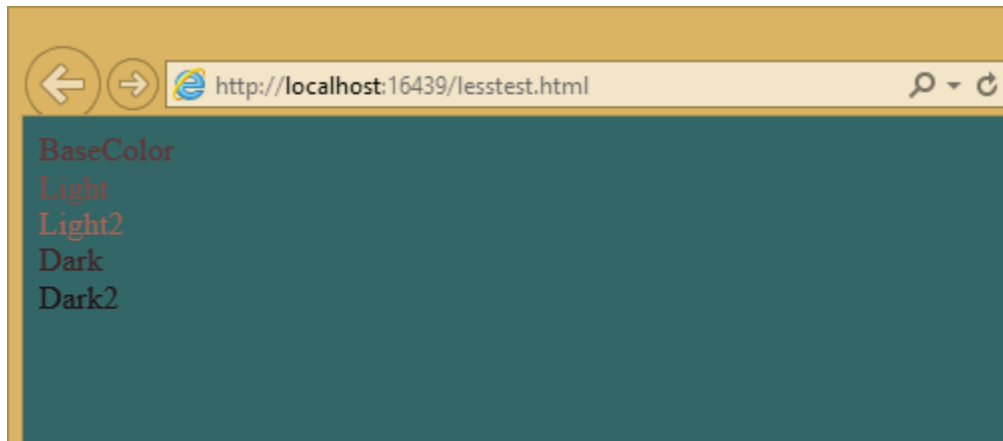
Less also provides support for nested rules, as well as nested media queries. For example, defining nested hierarchies like menus can result in verbose CSS rules like these:

```css
nav {
  height: 40px;
  width: 100%;
}
nav li {
  height: 38px;
  width: 100px;
}
nav li a:link {
  color: #000;
  text-decoration: none;
}
nav li a:visited {
  text-decoration: none;
  color: #CC3333;
}
nav li a:hover {
  text-decoration: underline;
```

```
    font-weight: bold;
}
nav li a:active {
  text-decoration: underline;
}
```

Ideally all of the related style rules will be placed together within the CSS file, but in practice there is nothing enforcing this rule except convention and perhaps block comments.

Defining these same rules using Less looks like this:

```
nav {
  height: 40px;
  width: 100%;
  li {
    height: 38px;
    width: 100px;
    a {
      color: #000;
      &:link { text-decoration:none}
      &:visited { color: #CC3333; text-decoration:none}
      &:hover { text-decoration:underline; font-weight:bold}
      &:active {text-decoration:underline}
    }
  }
}
```

Note that in this case, all of the subordinate elements of `nav` are contained within its scope. There is no longer any repetition of parent elements (`nav`, `li`, `a`), and the total line count has dropped as well (though some of that is a result of putting values on the same lines in the second example). It can be very helpful, organizationally, to see all of the rules for a given UI element within an explicitly bounded scope, in this case set off from the rest of the file by curly braces.

The `&` syntax is a Less selector feature, with `&` representing the current selector parent. So, within the a {...} block, `&` represents an `a` tag, and thus `&:link` is equivalent to `a:link`.

Media queries, extremely useful in creating responsive designs, can also contribute heavily to repetition and complexity in CSS. Less allows media queries to be nested within classes, so that the entire class definition doesn't need to be repeated within different top-level `@media` elements. For example, this CSS for a responsive menu:

```
.navigation {
  margin-top: 30%;
  width: 100%;
}
@media screen and (min-width: 40em) {
  .navigation {
    margin: 0;
  }
}
@media screen and (min-width: 62em) {
  .navigation {
    width: 960px;
    margin: 0;
  }
}
```

This can be better defined in Less as:

```
.navigation {
  margin-top: 30%;
  width: 100%;
  @media screen and (min-width: 40em) {
    margin: 0;
  }
  @media screen and (min-width: 62em) {
    width: 960px;
    margin: 0;
  }
}
```

Another feature of Less that we have already seen is its support for mathematical operations, allowing style attributes to be constructed from pre-defined variables. This makes updating related styles much easier, since the base variable can be modified and all dependent values change automatically.

CSS files, especially for large sites (and especially if media queries are being used), tend to get quite large over time, making working with them unwieldy. Less files can be defined separately, then pulled together using `@import` directives. Less can also be used to import individual CSS files, as well, if desired.

Mixins can accept parameters, and Less supports conditional logic in the form of mixin guards, which provide a declarative way to define when certain mixins take effect. A common use for mixin guards is to adjust colors based on how light or dark the source color is. Given a mixin that accepts a parameter for color, a mixin guard can be used to modify the mixin based on that color:

```
.box (@color) when (lightness(@color) >= 50%) {
  background-color: #000;
}
.box (@color) when (lightness(@color) < 50%) {
  background-color: #FFF;
}
.box (@color) {
  color: @color;
}

.feature {
  .box (@base);
}
```

Given our current `@base` value of `#663333`, this Less script will produce the following CSS:

```
.feature {
  background-color: #FFF;
  color: #663333;
}
```

Less provides a number of additional features, but this should give you some idea of the power of this preprocessing language.

Sass

Sass is similar to Less, providing support for many of the same features, but with slightly different syntax. It is built using Ruby, rather than JavaScript, and so has different setup requirements. The original Sass language did not use curly braces or semicolons, but instead defined scope using white space and indentation. In version 3 of Sass, a new syntax was introduced, **SCSS** ("Sassy CSS"). SCSS is similar to CSS in that it ignores indentation levels and whitespace, and instead uses semicolons and curly braces.

To install Sass, typically you would first install Ruby (pre-installed on Mac), and then run:

```
gem install sass
```

However, assuming you're running Visual Studio, you can get started with Sass in much the same way as you would with Less. Open package.json and add the "gulp-sass" package to `devDependencies`:

```
"devDependencies": {
  "gulp": "3.8.11",
  "gulp-less": "3.0.2",
  "gulp-sass": "1.3.3",
  "rimraf": "2.3.2"
}
```

Next, modify gulpfile.js to add a sass variable and a task to compile your Sass files and place the results in the wwwroot folder:

```
var gulp = require("gulp"),
  rimraf = require("rimraf"),
  fs = require("fs"),
  less = require("gulp-less"),
  sass = require("gulp-sass");

// other content removed

gulp.task("sass", function () {
  return gulp.src('Styles/main2.scss')
    .pipe(sass())
    .pipe(gulp.dest(project.webroot + '/css'));
});
```

Now you can add the Sass file main2.scss to the Styles folder in the root of the project:
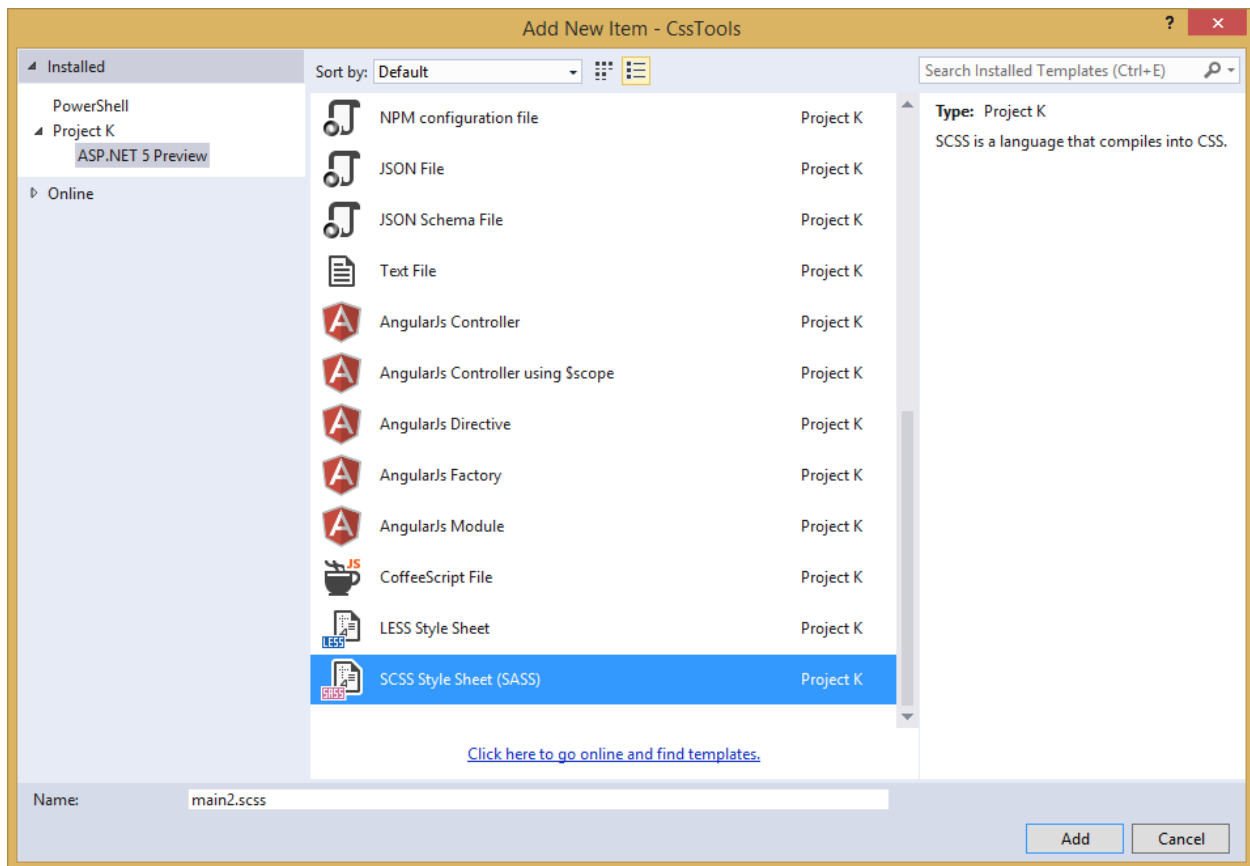
Open main2.scss and add the following:

```
$base: #CC0000;
body {
  background-color: $base;
}
```

Save all of your files. Now in Task Runner Explorer, you should see a sass task. Run it, refresh solution explorer, and look in the /wwwroot/css folder. There should be a main2.css file, with these contents:

```
body {
  background-color: #CC0000; }
```

Sass supports nesting in much the same was that Less does, providing similar benefits. Files can be split up by function and included using the @import directive:

```
@import 'anotherfile';
```

Sass supports mixins as well, using the @mixin keyword to define them and @include to include them, as in this example from sass-lang.com:

```
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
   -moz-border-radius: $radius;
    -ms-border-radius: $radius;
       border-radius: $radius;
}
```

```
.box { @include border-radius(10px); }
```

In addition to mixins, Sass also supports the concept of inheritance, allowing one class to extend another. It's conceptually similar to a mixin, but results in less CSS code. It's accomplished using the @extend keyword. First, let's see how we might use mixins, and the resulting CSS code. Add the following to your main2.scss file:

```scss
@mixin alert {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
}

.success {
  @include alert;
  border-color: green;
}

.error {
  @include alert;
  color: red;
  border-color: red;
  font-weight:bold;
}
```

Examine the output in main2.css after running the sass task in Task Runner Explorer:

```css
.success {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
  border-color: green;
 }

.error {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
  color: red;
  border-color: red;
  font-weight: bold;
}
```

Notice that all of the common properties of the alert mixin are repeated in each class. The mixin did a good job of helping use eliminate duplication at development time, but it's still creating CSS with a lot of duplication in it, resulting in larger than necessary CSS files - a potential performance issue. It would be great if we could follow the Don't Repeat Yourself (DRY) Principle at both development time and runtime.

Now replace the alert mixin with a .alert class, and change @include to @extend (remembering to extend .alert, not alert):

```scss
.alert {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
}

.success {
```

```
  @extend .alert;
  border-color: green;
}

.error {
  @extend .alert;
  color: red;
  border-color: red;
  font-weight:bold;
}
```

Run Sass once more, and examine the resulting CSS:

```
.alert, .success, .error {
  border: 1px solid black;
  padding: 5px;
  color: #333333; }

.success {
  border-color: green; }

.error {
  color: red;
  border-color: red;
  font-weight: bold; }
```

Now the properties are defined only as many times as needed, and better CSS is generated.

Sass also includes functions and conditional logic operations, similar to Less. In fact, the two languages' capabilities are very similar.

Less or Sass?

There is still no consensus as to whether it's generally better to use Less or Sass (or even whether to prefer the original Sass or the newer SCSS syntax within Sass). A recent poll conducted on twitter of mostly ASP.NET developers found that the majority preferred to use Less, by about a 2-to-1 margin. Probably the most important decision is to **use one of these tools**, as opposed to just hand-coding your CSS files. Once you've made that decision, both Less and Sass are good choices.

Font Awesome

In addition to CSS pre-compilers, another great resource for styling modern web applications is Font Awesome. Font Awesome is a toolkit that provides over 500 scalable vector icons that can be freely used in your web applications. It was originally designed to work with Bootstrap, but has no dependency on that framework, or on any JavaScript libraries.

The easiest way to get started with Font Awesome is to add a reference to it, using its public content delivery network (CDN) location:

```
<link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/font-awesome/4.3.0/css/font-awesome.min.css">
```

Of course, you can also quickly add it to your Visual Studio project by adding it to the "dependencies" in bower.json:

```
{
  "name": "ASP.NET",
  "private": true,
  "dependencies": {
    "bootstrap": "3.0.0",
    "jquery": "1.10.2",
    "jquery-validation": "1.11.1",
    "jquery-validation-unobtrusive": "3.2.2",
    "hammer.js": "2.0.4",
    "bootstrap-touch-carousel": "0.8.0",
    "Font-Awesome": "4.3.0"
  }
}
```

Then, to get the stylesheet added to the wwwroot folder, modify gulpfile.js as follows:

```
gulp.task("copy", ["clean"], function () {
  var bower = {
    "angular": "angular/angular*.{js,map}",
    "bootstrap": "bootstrap/dist/**/*.{js,map,css,ttf,svg,woff,eot}",
    "bootstrap-touch-carousel": "bootstrap-touch-carousel/dist/**/*.{js,css}",
    "hammer.js": "hammer.js/hammer*.{js,map}",
    "jquery": "jquery/jquery*.{js,map}",
    "jquery-validation": "jquery-validation/jquery.validate.js",
    "jquery-validation-unobtrusive": "jquery-validation-unobtrusive/jquery.validate.unobtrusive.js",
    "font-awesome": "Font-Awesome/**/*.{css,otf,eot,svg,ttf,woff,wof2}"
  };

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

Once this is in place (and saved), running the 'copy' task in Task Runner Explorer should copy the font awesome fonts and css files to `/lib/font-awesome`.

Once you have a reference to it on a page, you can add icons to your application by simply applying Font Awesome classes, typically prefixed with "fa-", to your inline HTML elements (such as `<span>` or `<i>`). As a very simple example, you can add icons to simple lists and menus using code like this:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
  <link href="lib/font-awesome/css/font-awesome.css" rel="stylesheet" />
</head>
<body>
  <ul class="fa-ul">
    <li><i class="fa fa-li fa-home"></i> Home</li>
    <li><i class="fa fa-li fa-cog"></i> Settings</li>
  </ul>
</body>
</html>
```

This produces the following in the browser - note the icon beside each item:
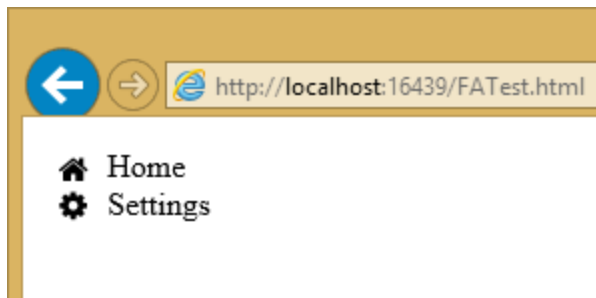
You can view a complete list of the available icons here:

http://fortawesome.github.io/Font-Awesome/icons/

Summary

Modern web applications increasingly demand responsive, fluid designs that are clean, intuitive, and easy to use from a variety of devices. Managing the complexity of the CSS stylesheets required to achieve these goals is best done using a pre-processor like Less or Sass. In addition, toolkits like Font Awesome quickly provide well-known icons to textual navigation menus and buttons, improving the overall user experience of your application.

1.8.8 Bundling and Minification

By Rick Anderson, Erik Reitan and Daniel Roth

Bundling and minification are two techniques you can use in ASP.NET to improve page load performance for your web application. Bundling combines multiple files into a single file. Minification performs a variety of different code optimizations to scripts and CSS, which results in smaller payloads. Used together, bundling and minification improves load time performance by reducing the number of requests to the server and reducing the size of the requested assets (such as CSS and JavaScript files).

This article explains the benefits of using bundling and minification, including how these features can be used with ASP.NET Core applications.

Sections:

- *Overview*
- *Bundling*
- *Minification*
- *Impact of Bundling and Minification*
- *Controlling Bundling and Minification*
- *See Also*

Overview

In ASP.NET Core apps, you bundle and minify the client-side resources during design-time using third party tools, such as *Gulp* and *Grunt*. By using design-time bundling and minification, the minified files are created prior to the application's deployment. Bundling and minifying before deployment provides the advantage of reduced server load. However, it's important to recognize that design-time bundling and minification increases build complexity and only works with static files.

Bundling and minification primarily improve the first page request load time. Once a web page has been requested, the browser caches the assets (JavaScript, CSS and images) so bundling and minification won't provide any performance

boost when requesting the same page, or pages on the same site requesting the same assets. If you don't set the expires header correctly on your assets, and you don't use bundling and minification, the browsers freshness heuristics will mark the assets stale after a few days and the browser will require a validation request for each asset. In this case, bundling and minification provide a performance increase even after the first page request.

Bundling

Bundling is a feature that makes it easy to combine or bundle multiple files into a single file. Because bundling combines multiple files into a single file, it reduces the number of requests to the server that is required to retrieve and display a web asset, such as a web page. You can create CSS, JavaScript and other bundles. Fewer files, means fewer HTTP requests from your browser to the server or from the service providing your application. This results in improved first page load performance.

Bundling can be accomplished using the gulp-concat plugin, which is installed with the Node Package Manager (npm). Add the `gulp-concat` package to the `devDependencies` section of your *package.json* file. To edit your *package.json* file from Visual Studio right-click on the **npm** node under **Dependencies** in the solution explorer and select **Open package.json**:

```
{
  "name": "asp.net",
  "version": "0.0.0",
  "private": true,
  "devDependencies": {
    "gulp": "3.8.11",
    "gulp-concat": "2.5.2",
    "gulp-cssmin": "0.1.7",
    "gulp-uglify": "1.2.0",
    "rimraf": "2.2.8"
  }
}
```

Run `npm install` to install the specified packages. Visual Studio will automatically install npm packages whenever *package.json* is modified.

In your *gulpfile.js* import the `gulp-concat` module:

```
var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");
```

Use globbing patterns to specify the files that you want to bundle and minify:

```
var paths = {
    js: webroot + "js/**/*.js",
    minJs: webroot + "js/**/*.min.js",
    css: webroot + "css/**/*.css",
    minCss: webroot + "css/**/*.min.css",
    concatJsDest: webroot + "js/site.min.js",
    concatCssDest: webroot + "css/site.min.css"
};
```

You can then define gulp tasks that run `concat` on the desired files and output the result to your webroot:

```
gulp.task("min:js", function () {
    return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
    return gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});
```

The gulp.src function emits a stream of files that can be piped to gulp plugins. An array of globs specifies the files to emit using node-glob syntax. The glob beginning with ! excludes matching files from the glob results up to that point.

Minification

Minification performs a variety of different code optimizations to reduce the size of requested assets (such as CSS, image, JavaScript files). Common results of minification include removing unnecessary white space and comments, and shortening variable names to one character.

Consider the following JavaScript function:

```
AddAltToImg = function (imageTagAndImageID, imageContext) {
  ///<signature>
  ///<summary> Adds an alt tab to the image
  // </summary>
  //<param name="imgElement" type="String">The image selector.</param>
  //<param name="ContextForImage" type="String">The image context.</param>
  ///</signature>
  var imageElement = $(imageTagAndImageID, imageContext);
  imageElement.attr('alt', imageElement.attr('id').replace(/ID/, ''));
}
```

After minification, the function is reduced to the following:

```
AddAltToImg=function(t,a){var r=$(t,a);r.attr("alt",r.attr("id").replace(/ID/,""))};
```

In addition to removing the comments and unnecessary whitespace, the following parameters and variable names were renamed (shortened) as follows:

Original	Renamed
imageTagAndImageID	t
imageContext	a
imageElement	r

To minify your JavaScript files you can use the gulp-uglify plugin. For CSS you can use the gulp-cssmin plugin. Install these packages using npm as before:

```
{
  "name": "asp.net",
  "version": "0.0.0",
  "private": true,
  "devDependencies": {
    "gulp": "3.8.11",
```

```
    "gulp-concat": "2.5.2",
    "gulp-cssmin": "0.1.7",
    "gulp-uglify": "1.2.0",
    "rimraf": "2.2.8"
  }
}
```

Import the `gulp-uglify` and `gulp-cssmin` modules in your *gulpfile.js* file:

```
var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");
```

Add `uglify` to minify your bundled JavaScript files and `cssmin` to minify your bundled CSS files.

```
gulp.task("min:js", function () {
    return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
    return gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});
```

To run bundling and minification tasks from the command-line using gulp (`gulp min`), or you can also execute any of your gulp tasks from within Visual Studio using the **Task Runner Explorer**. To use the **Task Runner Explorer** select *gulpfile.js* in the Solution Explorer and then select **Tools > Task Runner Explorer**:
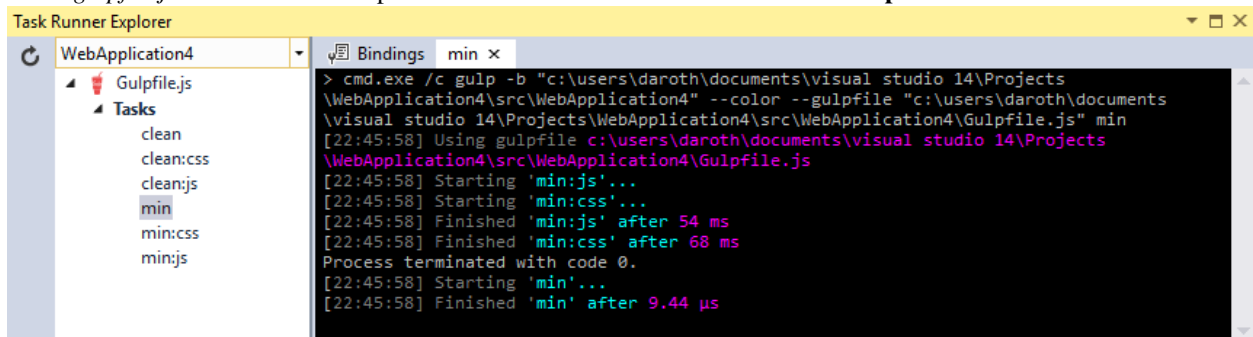
: The gulp tasks for bundling and minification do not general run when your project is built and must be run manually.

Impact of Bundling and Minification

The following table shows several important differences between listing all the assets individually and using bundling and minification on a simple web page:

Action	With B/M	Without B/M	Change
File Requests	7	18	157%
KB Transferred	156	264.68	70%
Load Time (MS)	885	2360	167%

The bytes sent had a significant reduction with bundling as browsers are fairly verbose with the HTTP headers that they apply on requests. The load time shows a big improvement, however this example was run locally. You will get greater gains in performance when using bundling and minification with assets transferred over a network.

Controlling Bundling and Minification

In general, you want to use the bundled and minified files of your app only in a production environment. During development, you want to use your original files so your app is easier to debug.

You can specify which scripts and CSS files to include in your pages using the environment tag helper in your layout pages(See *Tag Helpers*). The environment tag helper will only render its contents when running in specific environments. See *Working with Multiple Environments* for details on specifying the current environment.

The following environment tag will render the unprocessed CSS files when running in the `Development` environment:

```
1  <environment names="Development">
2      <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
3      <link rel="stylesheet" href="~/css/site.css" />
4  </environment>
```

This environment tag will render the bundled and minified CSS files only when running in `Production` or `Staging`:

```
1  <environment names="Staging,Production">
2      <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.css
3              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
4              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-v
5      <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
6  </environment>
```

See Also

- *Using Gulp*
- *Using Grunt*
- *Working with Multiple Environments*
- *Tag Helpers*

1.8.9 Working with a Content Delivery Network (CDN)

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.8.10 Responsive Design for the Mobile Web

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.8.11 Building Projects with Yeoman

By Steve Smith, Scott Addie, Rick Anderson and Noel Rice

Yeoman generates complete projects for a given set of client tools. Yeoman is an open-source tool that works like a Visual Studio project template. The Yeoman command line tool yo works alongside a Yeoman generator. Generators define the technologies that go into a project.

Sections:

- *Install Node.js, npm, and Yeoman*
- *Create an ASP.NET app*
- *Restore, build and run*
- *Specifying the client-side task runner*
- *Building and Running from Visual Studio*
- *Restoring, Building, and Hosting from the Command Line*
- *Adding to Your Project with Sub Generators*
- *Related Resources*

Install Node.js, npm, and Yeoman

To get started with Yeoman install Node.js. The installer includes Node.js and npm.

Follow the instructions on http://yeoman.io/learning/ to install yo, bower, grunt, and gulp.

```
npm install -g yo bower grunt-cli gulp
```

: If you get the error `npm ERR! Please try running this command again as root/Administrator.` on Mac OS, run the following command using sudo: `sudo npm install -g yo bower grunt-cli gulp`

From the command line, install the ASP.NET generator:

```
npm install -g generator-aspnet
```

: If you get a permission error, run the command under `sudo` as described above.

The `-g` flag installs the generator globally, so that it can be used from any path.

Create an ASP.NET app

Create a directory for your projects

```
mkdir src
cd src
```

Run the ASP.NET generator for `yo`
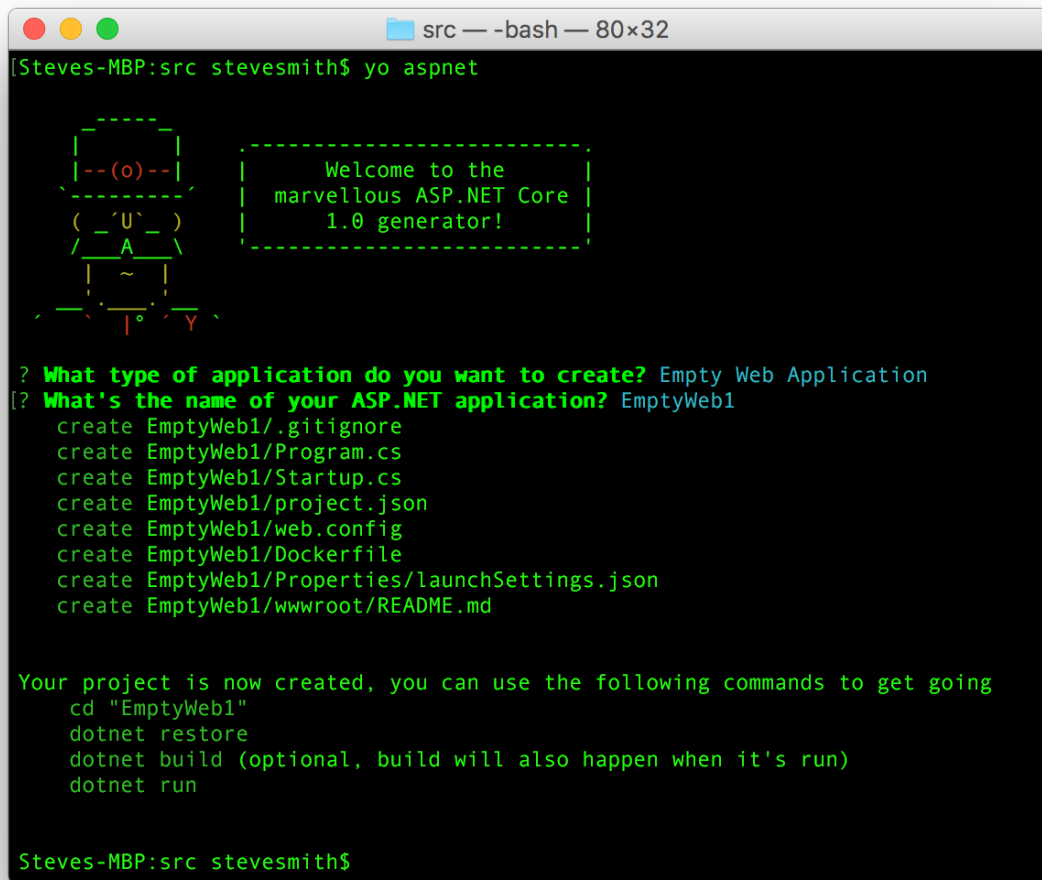
```
yo aspnet
```

The generator displays a menu. Arrow down to the **Empty Web Application** project and tap **Enter**:
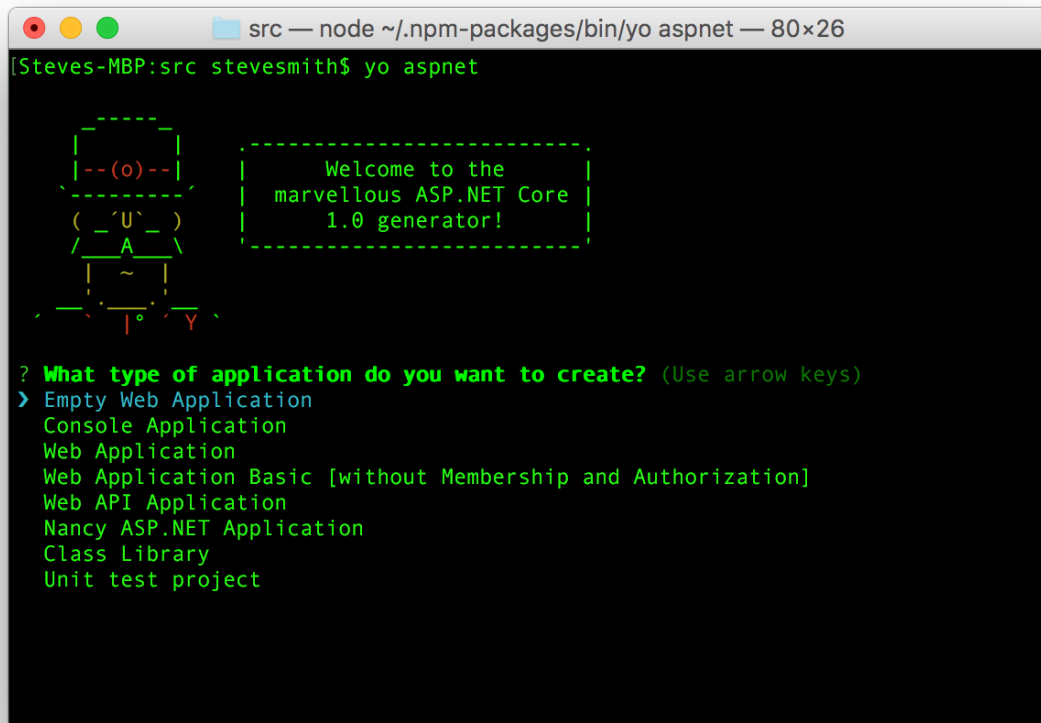
Use "EmptyWeb1" for the app name and then tap **Enter**

Yeoman will scaffold the project and its supporting files. Suggested next steps are also provided in the form of commands.

The ASP.NET generator creates ASP.NET Core projects that can be loaded into Visual Studio Code, Visual Studio, or run from the command line.

Restore, build and run

Follow the suggested commands by changing directories to the `EmptyWeb1` directory. Then run `dotnet restore`.

Build and run the app using `dotnet build` and `dotnet run`:

At this point you can navigate to the URL shown to test the newly created ASP.NET Core app.

: If you were directed to this tutorial from *Your First ASP.NET Core Application on a Mac Using Visual Studio Code*, you can return now.

Specifying the client-side task runner

The ASP.NET generator creates supporting files to configure client-side build tools. A *Grunt* or *Gulp* task runner file is added to your project to automate build tasks for Web projects. The default generator creates *gulpfile.js* to run tasks. Running the generator with the `--grunt` argument generates *Gruntfile.js*:

```
yo aspnet --grunt
```

The generator also configures *package.json* to load *Grunt* or *Gulp* dependencies. It also adds *bower.json* and *.bowerrc* files to restore client-side packages using the *Bower* client-side package manager.

Building and Running from Visual Studio

You can load your generated ASP.NET Core web project directly into Visual Studio, then build and run your project from there. Follow the instructions above to scaffold a new ASP.NET Core app using yeoman. This time, choose **Web Application** from the menu and name the app `MyWebApp`.

Open Visual Studio. From the File menu, select *Open → Project/Solution*.

In the Open Project dialog, navigate to the *project.json* file, select it, and click the **Open** button. In the Solution Explorer, the project should look something like the screenshot below.
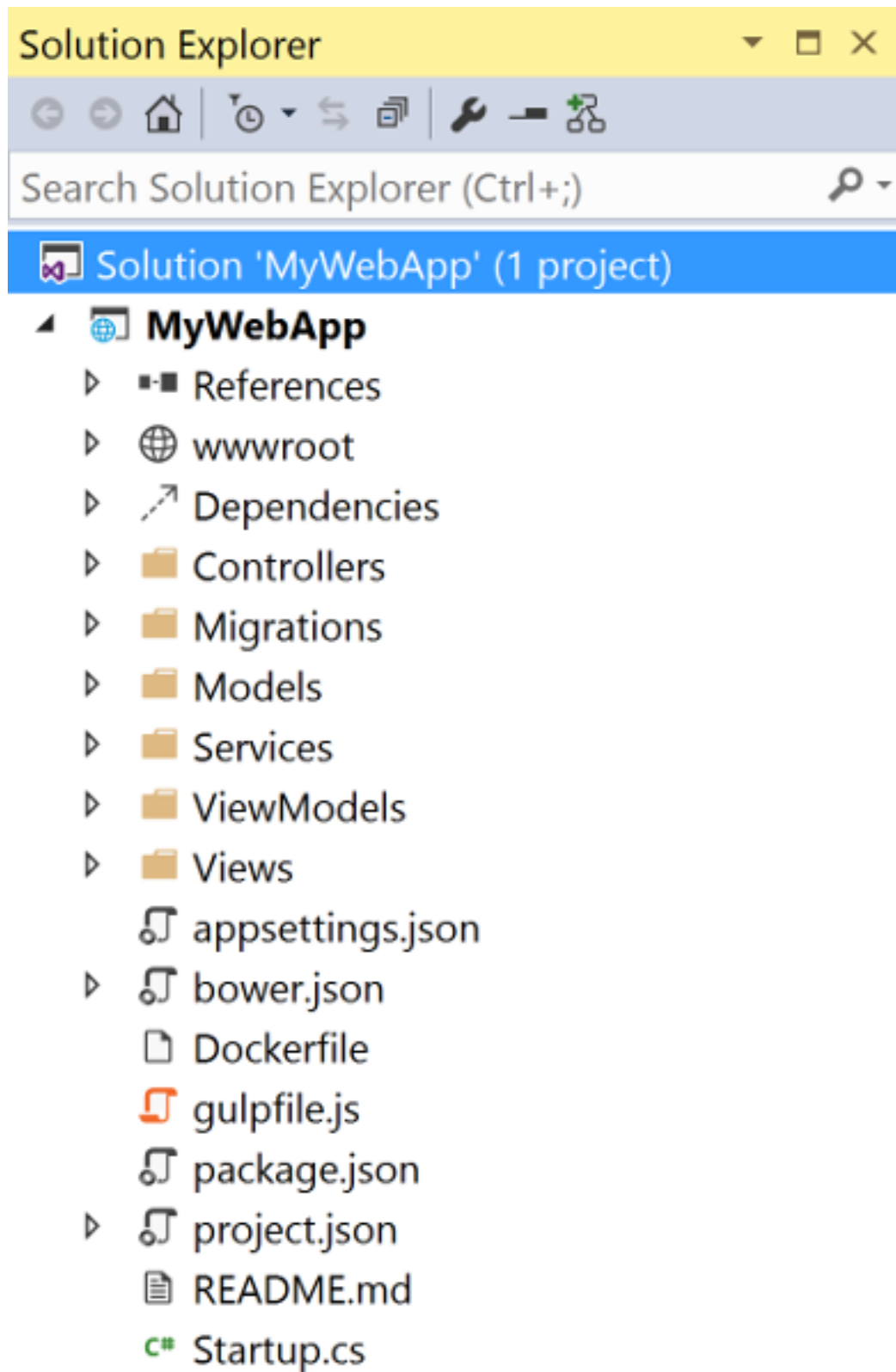
Yeoman scaffolds a MVC web application, complete with both server- and client-side build support. Server-side dependencies are listed under the **References** node, and client-side dependencies in the **Dependencies** node of Solution

Explorer. Dependencies are restored automatically when the project is loaded.
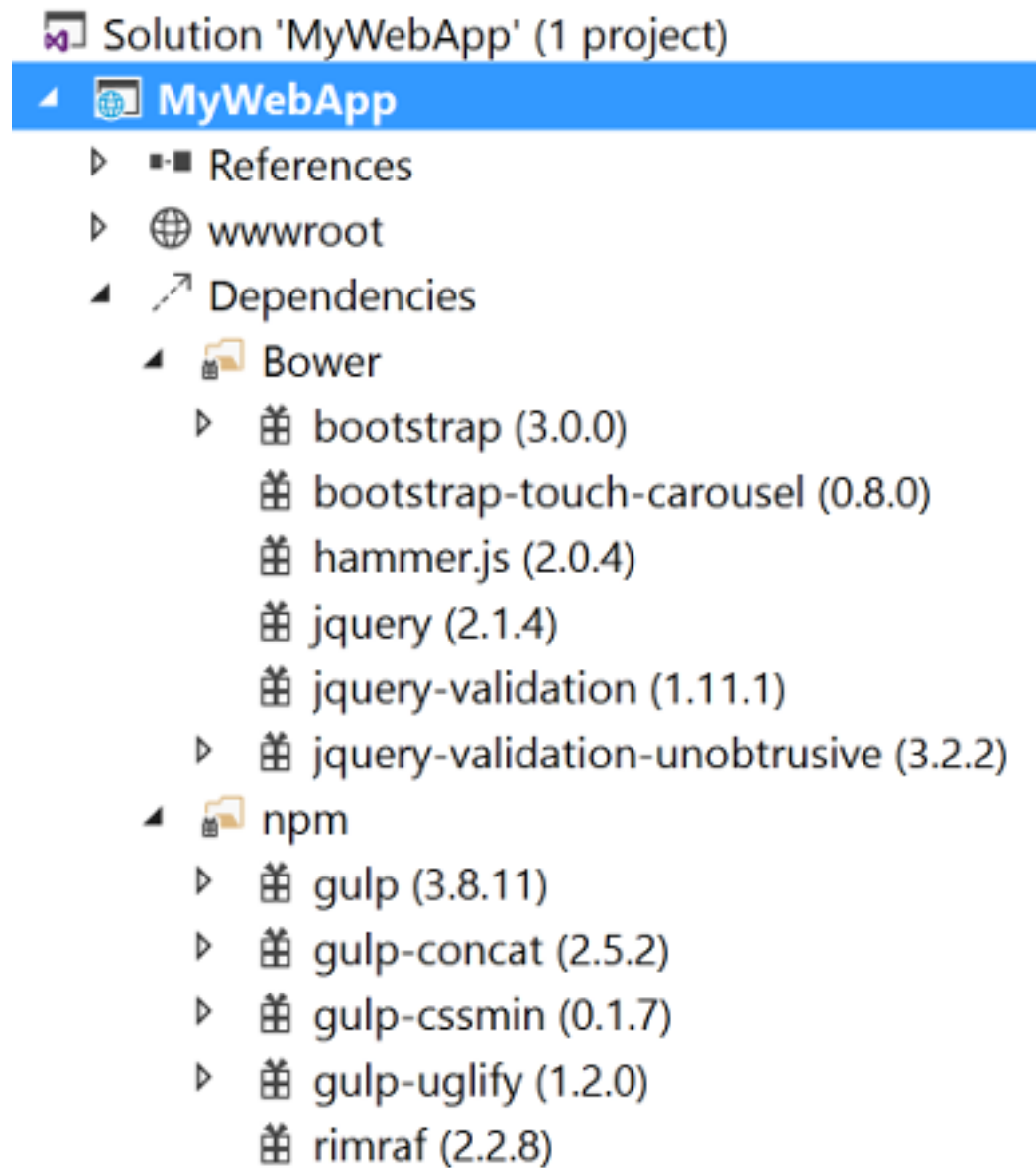
When all the dependencies are restored, press **F5** to run the project. The default home page displays in the browser.
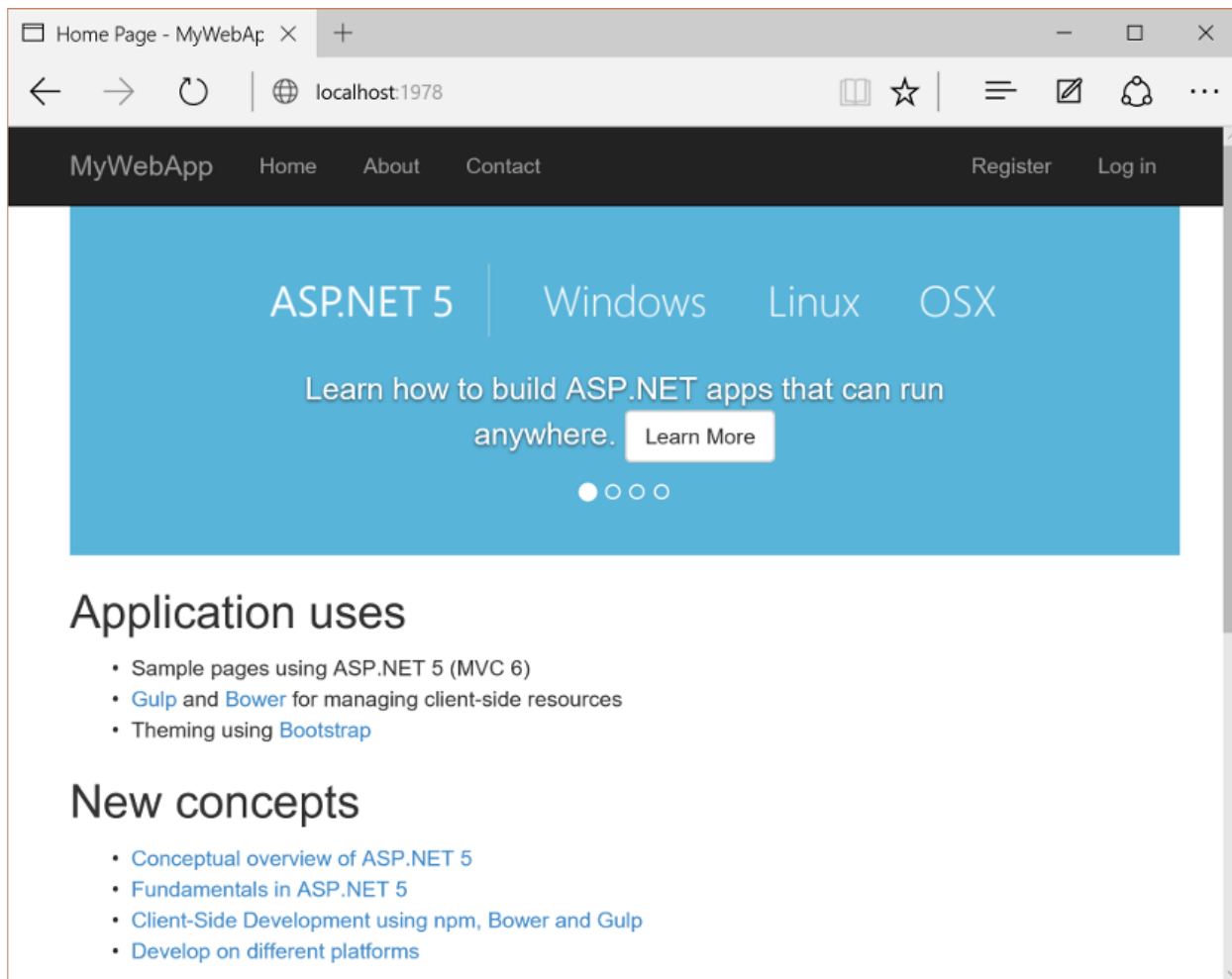
Restoring, Building, and Hosting from the Command Line

You can prepare and host your web application using the .NET Core command-line interface.

From the command line, change the current directory to the folder containing the project (that is, the folder containing the *project.json* file):

```
cd src\MyWebApp
```

From the command line, restore the project's NuGet package dependencies:

```
dotnet restore
```

Run the application:

```
dotnet run
```

The cross-platform *Kestrel* web server will begin listening on port 5000.

Open a web browser, and navigate to http://localhost:5000.
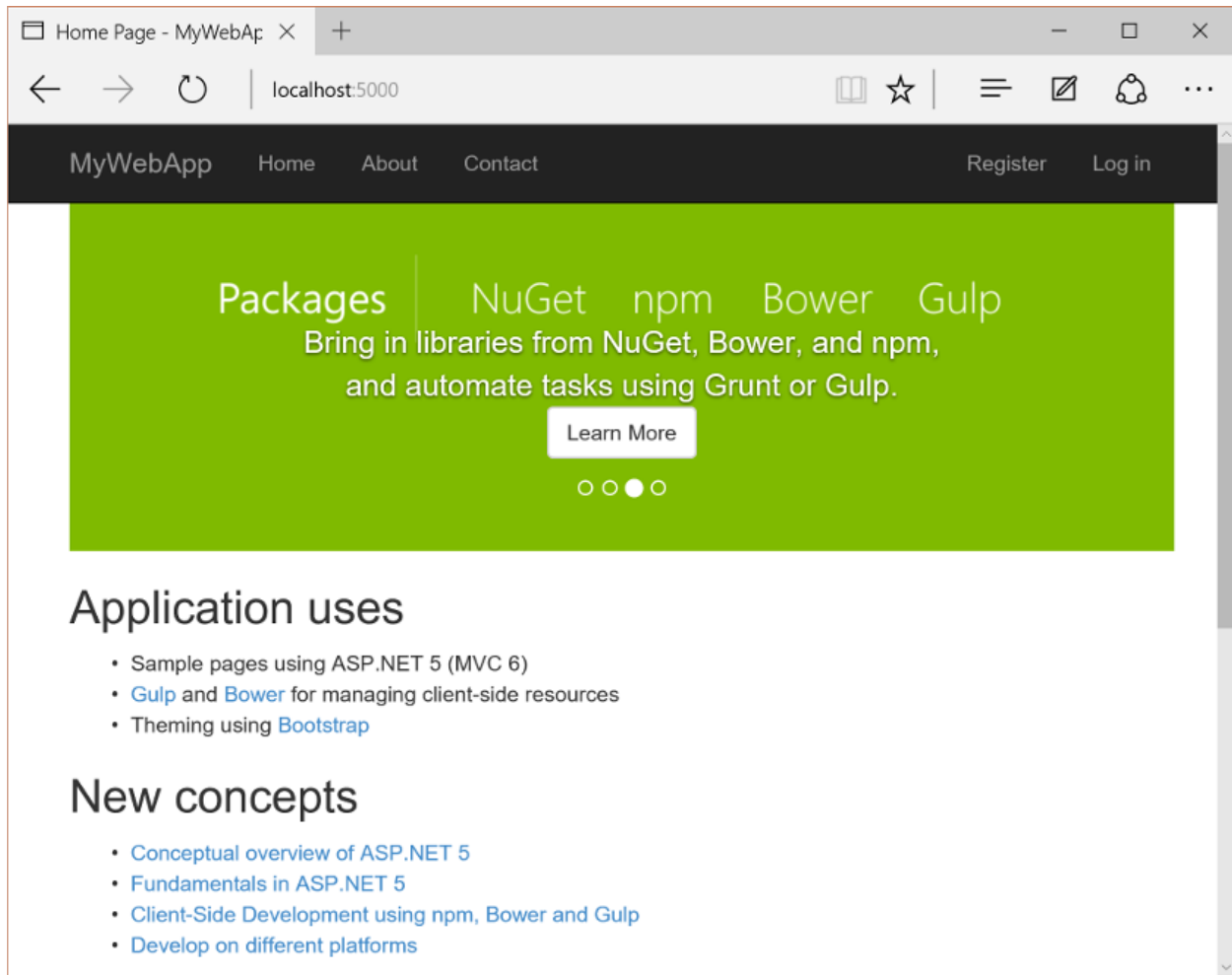
Adding to Your Project with Sub Generators

You can add new generated files using Yeoman even after the project is created. Use sub generators to add any of the file types that make up your project. For example, to add a new class to your project, enter the `yo aspnet:Class` command followed by the name of the class. Execute the following command from the directory in which the file should be created:

```
yo aspnet:Class Person
```

The result is a file named Person.cs with a class named `Person`:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace MyNamespace
{
    public class Person
    {
        public Person()
        {
```

```
            }
        }
}
```

Related Resources

- *Servers (HttpPlatformHandler, Kestrel and WebListener)*

- *Your First ASP.NET Core Application on a Mac Using Visual Studio Code*

- *Fundamentals*

1.9 Mobile

1.10 Publishing and Deployment

1.10.1 Publishing to IIS

By Luke Latham and Rick Anderson

> **Sections:**
>
> - *Supported operating systems*
> - *IIS configuration*
> - *Install the .NET Core Windows Server Hosting bundle*
> - *Application configuration*
> - *Deploy the application*
> - *Configure the website in IIS*
> - *Create a Data Protection Registry Hive*
> - *Common errors*
> - *Additional resources*

Supported operating systems

The following operating systems are supported:

- Windows 7 and newer

- Windows Server 2008 R2 and newer*

*Conceptually, the IIS configuration described in this document also applies to hosting ASP.NET Core applications on Nano Server IIS, but refer to *ASP.NET Core on Nano Server* for specific instructions.

IIS configuration

Enable the **Web Server (IIS)** server role and establish role services.

Windows desktop operating systems

Navigate to **Control Panel > Programs > Programs and Features > Turn Windows features on or off** (left side of the screen). Open the group for **Internet Information Services** and **Web Management Tools**. Check the box for **IIS Management Console**. Check the box for **World Wide Web Services**. Accept the default features for **World Wide Web Services** or customize the IIS features to suit your needs.
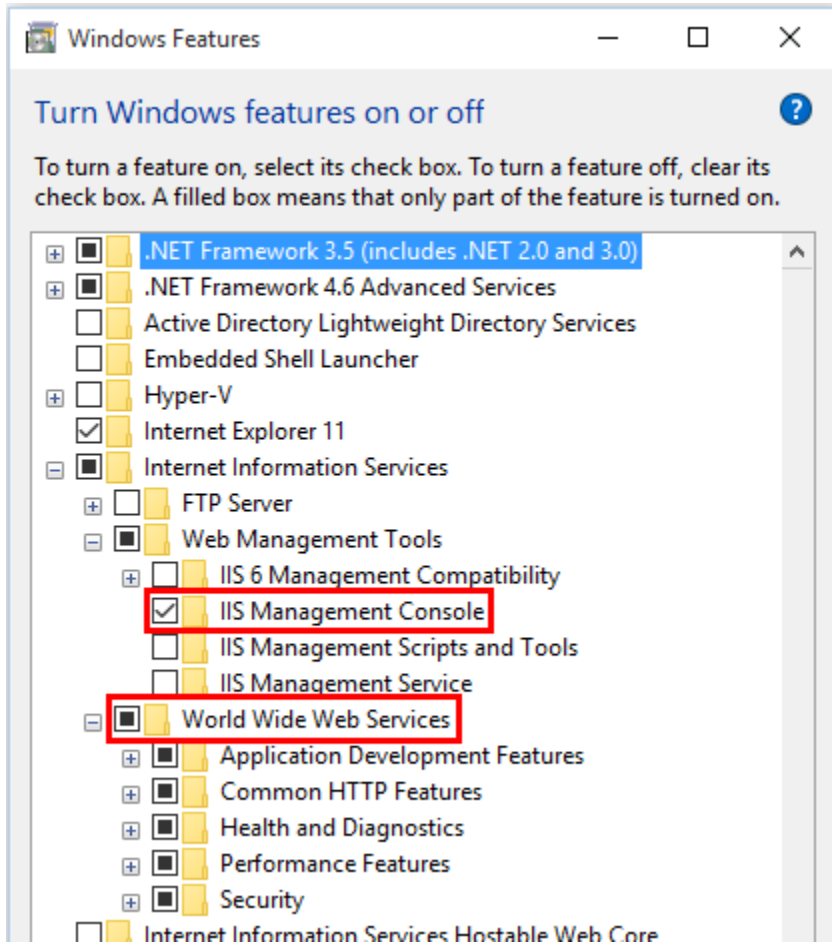
Windows Server operating systems

For server operating systems, use the **Add Roles and Features Wizard** via the **Manage** menu or the link in **Server Manager**. On the **Server Roles** step, check the box for **Web Server (IIS)**.
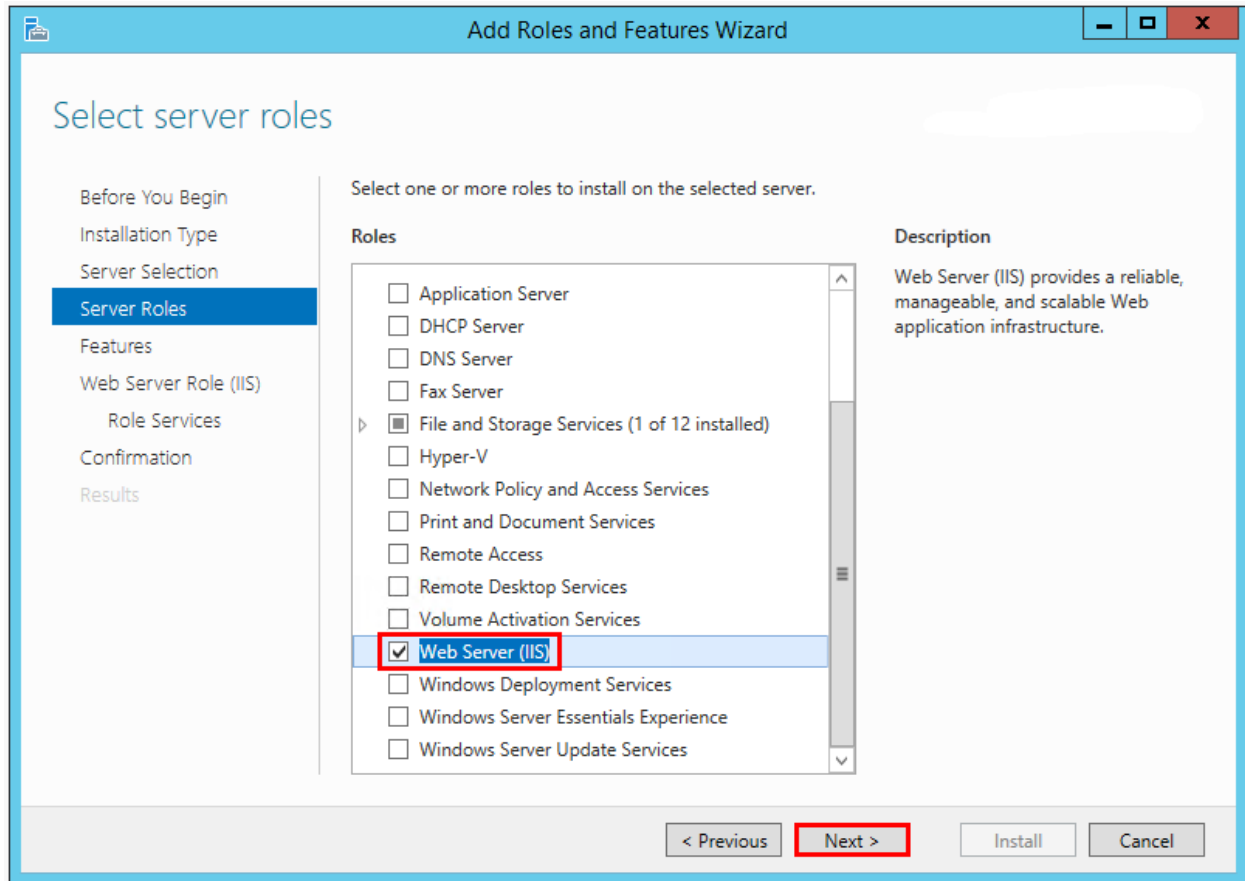
On the **Role services** step, select the IIS role services you desire or accept the default role services provided.
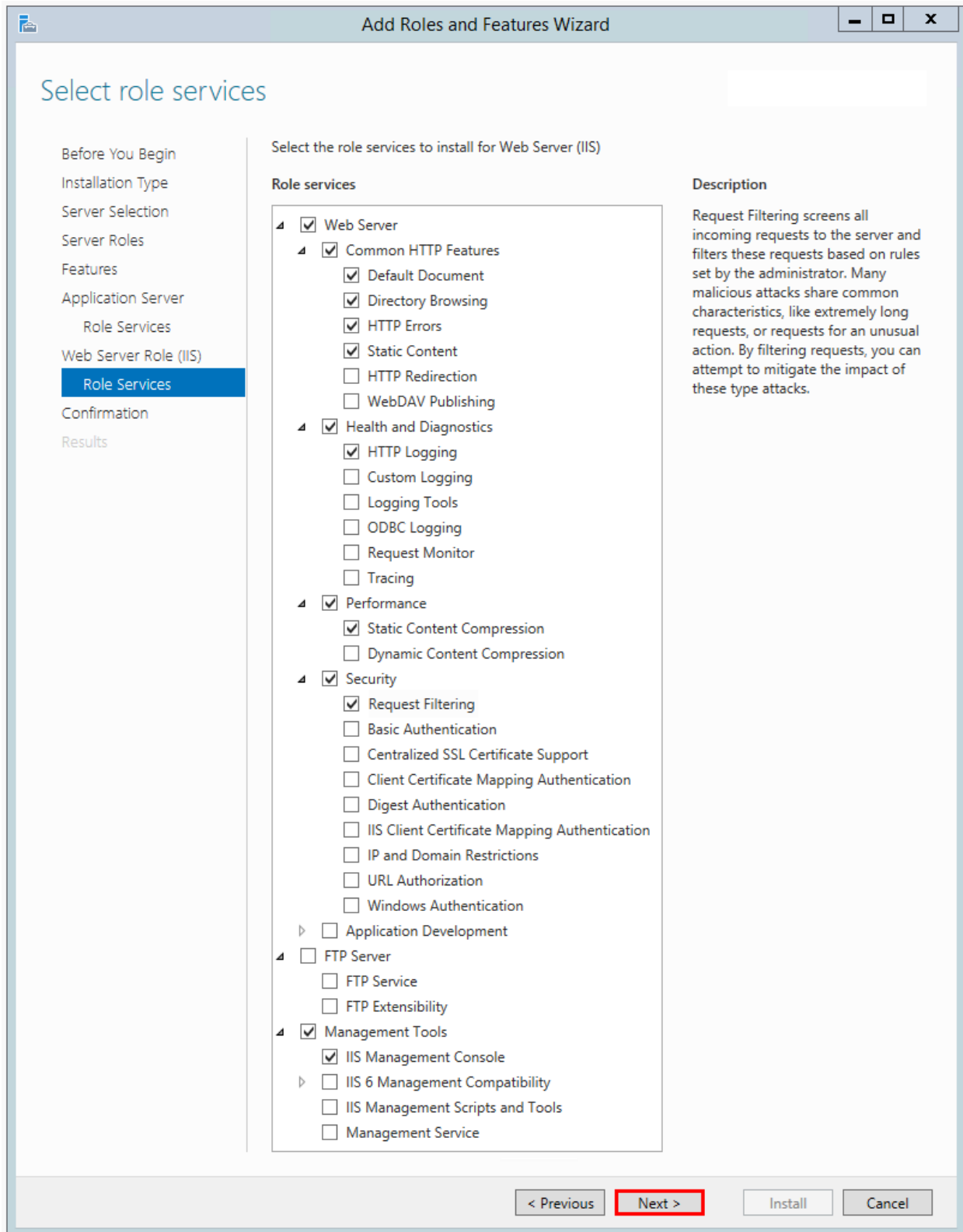
Proceed through the **Confirmation** step to enable the web server role and services.

Install the .NET Core Windows Server Hosting bundle

1. Install the .NET Core Windows Server Hosting bundle on the server. The bundle will install the .NET Core Runtime, .NET Core Library, and the ASP.NET Core Module. The module creates the reverse-proxy between IIS and the Kestrel server.

2. Execute **iisreset** at the command line or restart the server to pickup changes to the system PATH.

For more information on the ASP.NET Core Module, including configuration of the module and setting environment variables with *web.config*, the use of *app_offline.htm* to suspend request processing, and activation of module logging, see *ASP.NET Core Module Configuration Reference*.

Application configuration

Setting *IISOptions* for the *IISIntegration* service

To configure *IISIntegration* service options, include a service configuration for *IISOptions* in *ConfigureServices*.

```
services.Configure<IISOptions>(options => {
  ...
});
```

Option	Setting
AutomaticAuthentication	If true, the authentication middleware will alter the request user arriving and respond to generic challenges. If false, the authentication middleware will only provide identity and respond to challenges when explicitly indicated by the AuthenticationScheme.
ForwardClientCertificate	If true and the *MS-ASPNETCORE-CLIENTCERT* request header is present, the *ITLSConnectionFeature* will be populated.
ForwardWindowsAuthentication	If true, authentication middleware will attempt to authenticate using platform handler windows authentication. If false, authentication middleware won't be added.

publish-iis tool

The *publish-iis* tool can be added to any .NET Core application and will configure the ASP.NET Core Module by creating or modifying the *web.config* file. The tool runs after publishing with the *dotnet publish* command or publishing with Visual Studio and will configure the *processPath* and *arguments* for you. If you're publishing a *web.config* file

by including the file in your project and listing the file in the *publishOptions* section of *project.json*, the tool will not modify other IIS settings you have included in the file.

To include the *publish-iis* tool in your application, add entries to the *tools* and *scripts* sections of *project.json*.

```
"tools": {
  "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final"
  }
},
"scripts": {
  "postpublish": "dotnet publish-iis --publish-folder %publish:OutputPath% --framework %publish:Full]
}
```

Deploy the application

1. On the target IIS server, create a folder to contain the application's assets.

2. Within the folder you created, create a *logs* folder to hold application logs (if you plan to enable logging). If you plan to deploy your application with a *logs* folder in the payload, you may skip this step.

3. Deploy the application to the folder you created on the target IIS server. MSDeploy (Web Deploy) is the recommended mechanism for deployment, but you may use any of several methods to move the application to the server (for example, Xcopy, Robocopy, or PowerShell). Visual Studio users may use the default Visual Studio web publish script. For information on using Web Deploy, see *Publishing to IIS with Web Deploy using Visual Studio*.

> **:** .NET Core applications are hosted via a reverse-proxy between IIS and the Kestrel server. In order to create the reverse-proxy, the *web.config* file must be present at the content root path (typically the app base path) of the deployed application, which is the website physical path provided to IIS.
> Sensitive files exist on the app's physical path, including subfolders, such as *my_application.runtimeconfig.json*, *my_application.xml* (XML Documentation comments), and *my_application.deps.json*. The *web.config* file is required to create the reverse proxy to Kestrel, which prevents IIS from serving these and other sensitive files. **Therefore, it is important that the web.config file is never accidently renamed or removed from the deployment.**

Configure the website in IIS

1. In **IIS Manager**, create a new website. Provide a **Site name** and set the **Physical path** to the application's assets folder that you created. Provide the **Binding** configuration and create the website.

2. Set the application pool to **No Managed Code**. ASP.NET Core runs in a separate process and manages the runtime.

: If you change the default identity of the application pool from **ApplicationPoolIdentity**, verify the new identity has the required permissions to access the application's assets and database.

Open the **Add Website** window.

Configure the website.

In the **Application Pools** panel, open the **Edit Application Pool** window by right-clicking on the website's application pool and selecting **Basic Settings...** from the popup menu.
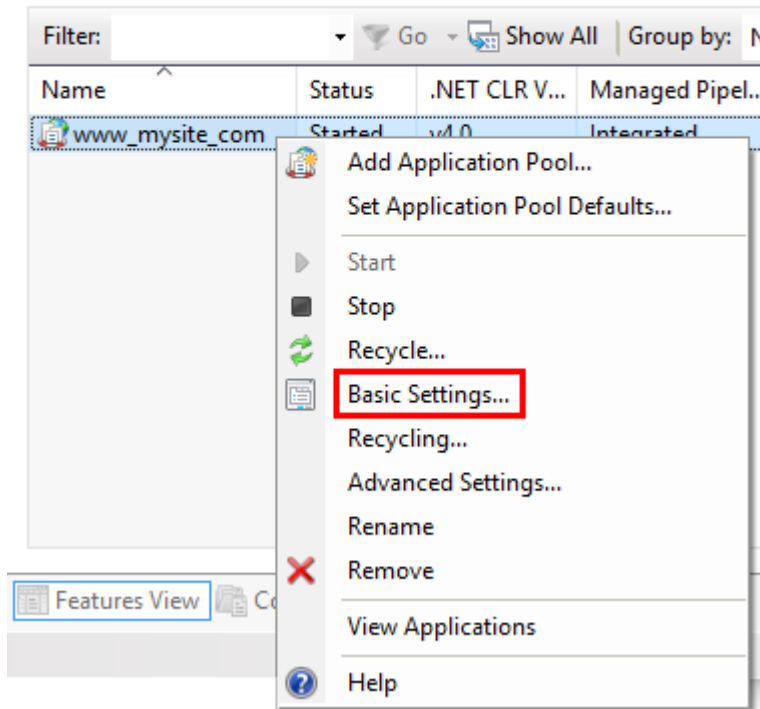
Set the **.NET CLR version** to **No Managed Code**.
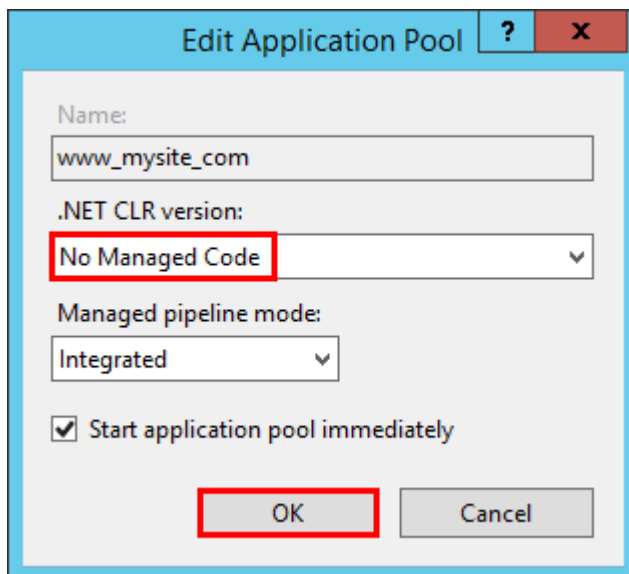
Browse the website.
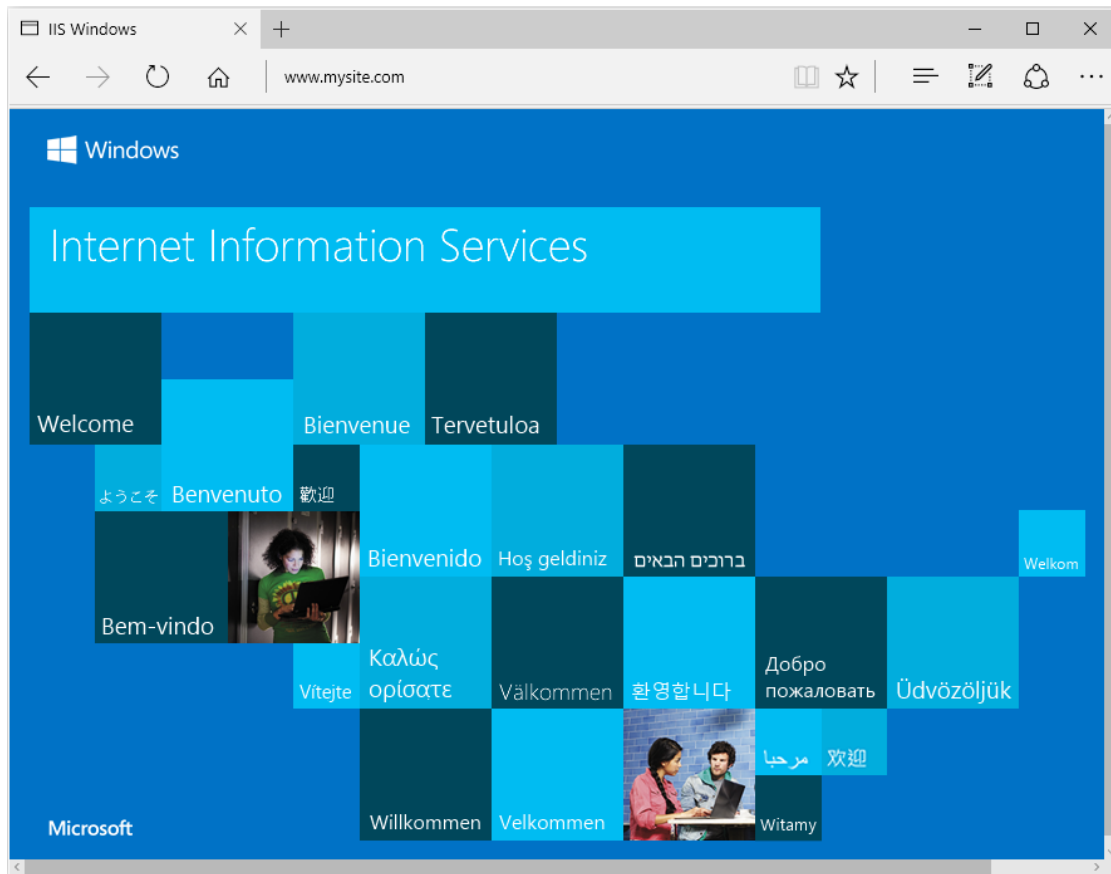
Create a Data Protection Registry Hive

Data Protection keys used by ASP.NET applications are stored in registry hives external to the applications. To persist the keys for a given application, you must create a registry hive for the application's application pool.

For standalone IIS installations, you may use the Data Protection Provision-AutoGenKeys.ps1 PowerShell script for each application pool used with an ASP.NET Core application. The keys will be persisted in the registry.

In web farm scenarios, an application can be configured to use a UNC path to store its data protection key ring. By default, the data protection keys are not encrypted. You can deploy an x509 certificate to each machine to encrypt the key ring. See *Configuring Data Protection* for details.

> **:** Data Protection is used by various ASP.NET middlewares, including those used in authentication. Even if you do not specifically call any Data Protection APIs from your own code you should configure Data Protection with the deployment script or in your own code. If you do not configure data protection when using IIS by default the keys will be held in memory and discarded when your application closes or restarts. This will then, for example, invalidate any cookies written by the cookie authentication and users will have to login again.

Common errors

The following is not a complete list of errors. Should you encounter an error not listed here, please leave a detailed error message in the DISQUS section below (click **Show comments** to open the DISQUS panel).

To diagnose problems with IIS deployments, study browser output, examine the server's **Application** log through **Event Viewer**, and enable module logging. The **ASP.NET Core Module** log will be found on the path provided in the *stdoutLogFile* attribute of the *<aspNetCore>* element in *web.config*. Any folders on the path provided in the attribute value must exist in the deployment. You must also set *stdoutLogEnabled="true"* to enable module logging. Applications that use the *publish-iis* tooling to create the *web.config* file will default the *stdoutLogEnabled* setting to *false*, so you must manually provide the file or modify the file in order to enable module logging.

A quick way to determine if the application is working properly is to run the application directly on Kestrel. If the application was published as a portable app, execute *dotnet <my_app>.dll* in the deployment folder. If the application was published as a self-contained app, run the application's executable directly on the command line, *<my_app>.exe*, in the deployment folder. If Kestrel is listening on default port 5000, you should be able to browse the application at *http://localhost:5000/*. If the application responds normally at the Kestrel endpoint address, the problem is more likely related to the IIS-ASP.NET Core Module-Kestrel configuration and less likely within the application itself.

A way to determine if the IIS reverse proxy to the Kestrel server is working properly is to perform a simple static file request for a stylesheet, script, or image from the application's static assets in *wwwroot* using *Static File middleware*. If the application can serve static files but MVC Views and other endpoints are failing, the problem is less likely related to the IIS-ASP.NET Core Module-Kestrel configuration and more likely within the application itself (for example, MVC routing or 500 Internal Server Error).

In most cases, enabling application logging will assist in troubleshooting problems with application or the reverse proxy. See *Logging* for more information.

Common errors and general troubleshooting instructions:

Installer unable to obtain VC++ Redistributable

- **Installer Exception:** Installation of the .NET Core Windows Server Hosting Bundle fails with *0x80070002 - The system cannot find the file specified*.

Troubleshooting:

- If the server does not have Internet access while installing the server hosting bundle, this exception will ensue when the installer is prevented from obtaining the *Microsoft Visual C++ 2015 Redistributable (x64)* packages online. You may obtain an installer for the packages from the Microsoft Download Center.

UseUrls called before UseIISIntegration

- **Browser:** No response

- **Application Log:** Process 'PROC_ID' failed to start. Port = PORT, Error Code = '-2147023829'.

- **ASP.NET Core Module Log:** Unhandled Exception: System.AggregateException: One or more errors occurred. (Error -4092 EACCES permission denied)

Troubleshooting:

- If your application uses the *UseUrls* extension on *WebHostBuilder*, make sure you have positioned the *UseUrls* extension before the *UseIISIntegration* extension on *WebHostBuilder*. *UseIISIntegration* must overwrite any values you provide in *UseUrls* in order for the reverse-proxy to succeed.

Platform conflicts with RID

- **Browser:** No response

- **Application Log:** Faulting module: KERNELBASE.dll Exception code: 0xe0434352 Faulting module path: C:WINDOWSsystem32KERNELBASE.dll

---

- **ASP.NET Core Module Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly 'teststandalone.dll' or one of its dependencies. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- If you published a self-contained application, confirm that you didn't set a **platform** in **buildOptions** of *project.json* that conflicts with the publishing RID. For example, do not specify a **platform** of **x86** and publish with an RID of **win81-x64** (**dotnet publish -c Release -r win81-x64**). The project will publish without warning or error but fail with the above logged exceptions on the server.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED

- **Application Log:** No entry

- **ASP.NET Core Module Log:** Log file not created

Troubleshooting:

- Confirm you are using the correct URI endpoint for the application. Check your bindings.

- Confirm that the IIS website is not in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

- **OS Exception:** The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the Microsoft HTTP Platform Handler 1.x.

Troubleshooting:

- Confirm that you have enabled the proper server role and features. See *IIS Configuration*.

Incorrect website physical path or application missing

- **Browser:** 403 Forbidden: Access is denied **–OR–** 403.14 Forbidden: The Web server is configured to not list the contents of this directory.

- **Application Log:** No entry

- **ASP.NET Core Module Log:** Log file not created

Troubleshooting:

- Check the IIS website **Basic Settings** and the physical application assets folder. Confirm that the application is in the folder at the IIS website **Physical path**.

Incorrect server role, module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error: The requested page cannot be accessed because the related configuration data for the page is invalid.

- **Application Log:** No entry

- **ASP.NET Core Module Log:** Log file not created

Troubleshooting:

- Confirm that you have enabled the proper server role. See *IIS Configuration*.

- Check **Programs & Features** and confirm that the **Microsoft ASP.NET Core Module** has been installed. If the **Microsoft ASP.NET Core Module** is not present in the list of installed programs, install the module. See *IIS Configuration*.

- Make sure that the **Application Pool Process Model Identity** is either set to **ApplicationPoolIdentity**; or if a custom identity is in use, confirm the identity has the correct permissions to access the application's assets folder.

Incorrect *proecessPath*, bundle not installed, server not restarted, or missing PATH var

- **Browser:** 502.3 Bad Gateway: There was a connection error while trying to route the request.

- **Application Log:** Process '0' failed to start. Port = PORT, Error Code = '-2147024894'.

- **ASP.NET Core Module Log:** Log file created but empty

Troubleshooting:

- Check the *processPath* attribute on the *<aspNetCore>* element in *web.config* to confirm that it is *dotnet* for a portable application or *.\my_application.exe* for a self-contained application.

- You may have deployed a portable application without installing .NET Core on the server. If you are attempting to deploy a portable application and have not installed .NET Core, run the **.NET Core Windows Server Hosting Bundle Installer** on the server. See *Install the .NET Core Windows Server Hosting Bundle*.

- You may have deployed a portable application and installed .NET Core without restarting the server. Restart the server.

- *dotnet.exe* might not accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.

Incorrect *arguments* of *<aspNetCore>* element

- **Browser:** 502.3 Bad Gateway: There was a connection error while trying to route the request.

- **Application Log:** Process 'PROC_ID' failed to start. Port = PORT, Error Code = '-2147023829'.

- **ASP.NET Core Module Log:** Unhandled Exception: System.FormatException: Unrecognized argument format **–OR –** Expected to load required hostpolicy.dll from [IIS_WEBSITE_PHYSICAL_PATH] - This may be because of an invalid .NET Core FX configuration in the directory.

Troubleshooting:

- Examine the *arguments* attribute on the *<aspNetCore>* element in *web.config* to confirm that it is either (a) *.\my_applciation.dll* for a portable application; or (b) not present, an empty string (*arguments=""*), or a list of your application's arguments (*arguments="arg1, arg2, ..."*) for a self-contained application.

Missing .NET Framework version

- **Browser:** 502.3 Bad Gateway: There was a connection error while trying to route the request.

- **Application Log:** Failed to start process with commandline '[IIS_WEBSITE_PHYSICAL_PATH] ', Error Code = '0x80004005'.

- **ASP.NET Core Module Log:** Missing method, file, or assembly exception. The method, file, or assembly specified in the exception is a .NET Framework method, file, or assembly.

Troubleshooting:

- Install the .NET Framework version missing from the server.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module Log:** Log file not created

Troubleshooting

- Confirm that the Application Pool is not in the *Stopped* state.

Additional resources

- *Introduction to ASP.NET Core*
- The Official Microsoft IIS Site
- Microsoft TechNet Library: Windows Server

1.10.2 Publishing to IIS with Web Deploy using Visual Studio

By Sayed Ibrahim Hashimi

Publishing an ASP.NET Core project to an IIS server with Web Deploy requires a few additional steps in comparison to an ASP.NET 4 project. We are working on simplifying the experience for the next version. Until then you can use these instructions to get started with publishing an ASP.NET Core web application using Web Deploy to any IIS host.

To publish an ASP.NET Core application to a remote IIS server the following steps are required.

1. Configure your remote IIS server to support ASP.NET Core
2. Create a publish profile
3. Customize the profile to support Web Deploy publish

In this document we will walk through each step.

Preparing your web server for ASP.NET Core

The first step is to ensure that your remote server is configured for ASP.NET Core. At a high level you'll need.

1. An IIS server with IIS 7.5+
2. Install HttpPlatformHandler
3. Install Web Deploy v3.6

The HttpPlatformHandler is a new component that connects IIS with your ASP.NET Core application. You can get that from the following download links.

- 64 bit HttpPlatformHandler
- 32 bit HttpPlatformHandler

In addition to installing the HttpPlatformHandler, you'll need to install the latest version of Web Deploy (version 3.6). To install Web Deploy 3.6 you can use the the Web Platform Installer. (WebPI) or directly from the download center. The preferred method is to use WebPI. WebPI offers a standalone setup as well as a configuration for hosting providers.

Configure Data Protection

To persist Data Protection keys you must create registry hives for each application pool to store the keys. You should use the Provisioning PowerShell script for each application pool you will be hosting ASP.NET Core applications under.

For web farm scenarios developers can configure their applications to use a UNC path to store the data protection key ring. By default this does not encrypt the key ring. You can deploy an x509 certificate to each machine and use that to encrypt the keyring. See the *configuration APIs* for more details.

> **:** Data Protection is used by various ASP.NET middlewares, including those used in authentication. Even if you do not specifically call any Data Protection APIs from your own code you should configure Data Protection with the deployment script or in your own code. If you do not configure data protection when using IIS by default the keys will be held in memory and discarded when your application closes or restarts. This will then, for example, invalidate any cookies written by the cookie authentication and users will have to login again.

You can find more info on configuring your IIS server for ASP.NET Core at *Publishing to IIS*. Now let's move on to the Visual Studio experience.

Publishing with Visual Studio

After you have configured your web server, the next thing to do is to create a publish profile in Visual Studio. The easiest way to get started with publishing an ASP.NET Core application to a standard IIS host is to use a publish profile. If your hosting provider has support for creating a publish profile, download that and then import it into the Visual Studio publish dialog with the Import button. You can see that dialog shown below.
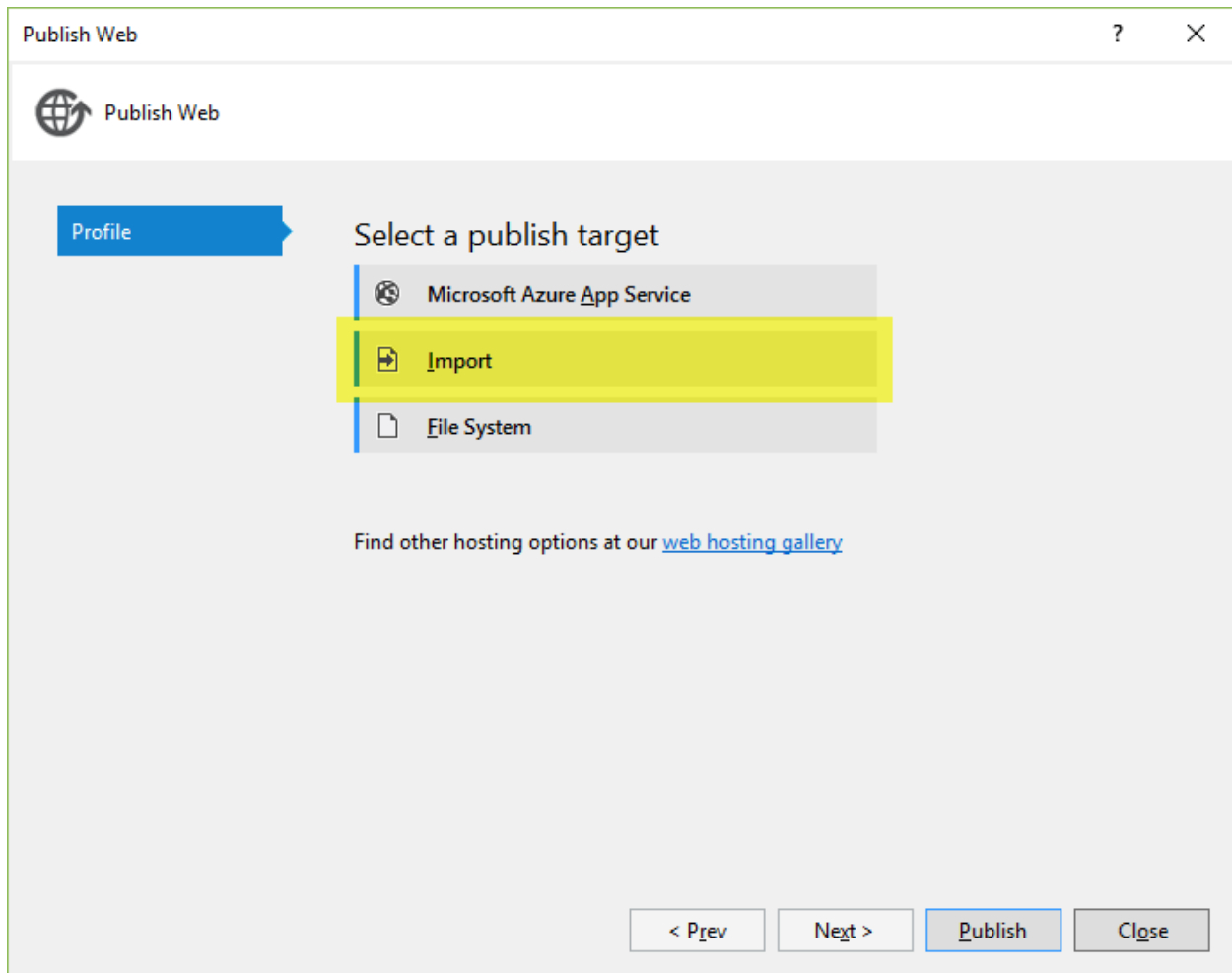
After importing the publish profile, there is one additional step that needs to be taken before being able to publish to a standard IIS host. In the publish PowerShell script generated (under PropertiesPublishProfiles) update the publish module version number from `1.0.1` to `1.0.2-beta2`. After changing `1.0.1` to `1.0.2-beta2` you can use the Visual Studio publish dialog to publish and preview changes.

1.10.3 How Web Publishing In Visual Studio Works

By Sayed Ibrahim Hashimi

The web publish experience for ASP.NET Core projects has significantly changed from ASP.NET 4. This doc will provide an overview of the changes and instructions on how to customize the publish process. Unless stated otherwise, the instructions in this article are for publishing from Visual Studio. For an overview of how to publish a web app on ASP.NET Core see *Publishing and Deployment*.

In ASP.NET when you publish a Visual Studio web project MSBuild is used to drive the entire process. The project file (.csproj or .vbproj) is used to gather the files that need to be published as well as perform any updates during publish (for example updating web.config). The project file contains the full list of files as well as their type. That information is used to determine the files to publish. The logic was implemented in an MSBuild .targets file. During the publish process MSBuild will call Web Deploy (msdeploy.exe) to transfer the files to the final location. To customize the publish process you would need to update the publish profile, or project file, with custom MSBuild elements.

In ASP.NET Core the publish process has been simplified, we no longer store a reference to files that the project contains. All files are included in the project by default (files can be excluded from the project or from publish by

updating *project.json*). When you publish an ASP.NET Core project from Visual Studio the following happens:

1. A publish profile is created at `Properties\PublishProfiles\profilename.pubxml`. The publish profile is an MSBuild file.

2. A PowerShell script is created at `Properties\PublishProfiles\profilename.ps1`.

3. `dotnet publish` is called to gather the files to publish to a temporary folder.

4. A PowerShell script is called passing in the properties from the publish profile and the location where `dotnet publish` has placed the files to publish.

To create a publish profile in Visual Studio, right click on the project in Solution Explorer and then select Publish.

The following image shows a visualization of this process.
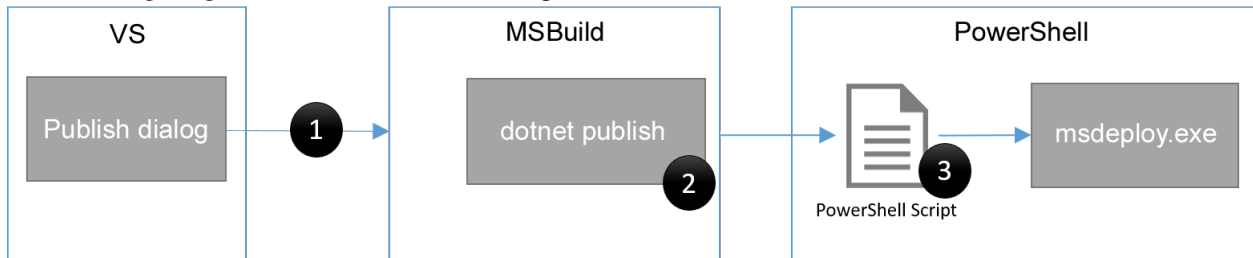
In the image above each black circle indicates an extension point, we will cover each extension point later in this document.

When you start a publish operation, the publish dialog is closed and then MSBuild is called to start the process. Visual Studio calls MSBuild to do this so that you can have parity with publishing when using Visual Studio or the command line. The MSBuild layer is pretty thin, for the most part it just calls `dotnet publish`. Let's take a closer look at `dotnet publish`.

The `dotnet publish` command will inspect project.json and the project folder to determine the files which need to be published. It will place the files needed to run the application in a single folder ready to be transferred to the final destination.

After `dotnet publish` has completed, the PowerShell script for the publish profile is called. Now that we have briefly discussed how publishing works at a high level let's take a look at the structure of the PowerShell script created for publishing.

When you create a publish profile in Visual Studio for an ASP.NET Core project a PowerShell script is created that has the following structure.

```powershell
[cmdletbinding(SupportsShouldProcess=$true)]
param($publishProperties=@{}, $packOutput,$pubProfilePath, $nugetUrl)

$publishModuleVersion = '1.0.2-beta2'

# functions to bootstrap the process when Visual Studio is not installed
# have been removed to simplify this doc

try{
    if (!(Enable-PublishModule)){
        Enable-PackageDownloader
        Enable-NuGetModule -name 'publish-module' -version $publishModuleVersion -nugetUrl $nugetUrl
    }

    'Calling Publish-AspNet' | Write-Verbose
    # call Publish-AspNet to perform the publish operation
    Publish-AspNet -publishProperties $publishProperties -packOutput $packOutput -pubProfilePath $pu
```

```
}
catch{
    "An error occurred during publish.n{0}" -f $_.Exception.Message | Write-Error
}
```

In the above snippet some functions have been removed for readability. Those functions are used to bootstrap the script in the case that it's executed from a machine which doesn't have Visual Studio installed. The script contains the following important elements:

1. Script parameters

2. Publish module version

3. Call to Publish-AspNet

The parameters of the script define the contract between Visual Studio and the PowerShell script. You should not change the declared parameters because Visual Studio depends on those. You can add additional parameters, but they must be added at the end.

The publish module version, denoted by `$publishModuleVersion`, defines the version of the web publish module that will be used. Valid version numbers can be found from published versions of the publish-module NuGet package on nuget.org. Once you create a publish profile the script definition is locked to a particular version of the publish-module package. If you need to update the version of the script you can delete the .ps1 file and then publish again in Visual Studio to get a new script created.

The call to Publish-AspNet moves the files from your local machine to the final destination. Publish-AspNet will be passed all the properties defined in the .pubxml file, even custom properties. For Web Deploy publish, msdeploy.exe will be called to publish the files to the destination. Publish-AspNet is passed the same parameters as the original script. You can get more info on the parameters for Publish-AspNet use Get-Help Publish-AspNet. If you get an error that the publish-module is not loaded, you can load it with

```
Import-Module "${env:ProgramFiles(x86)}\Microsoft Visual Studio 14.0\Common7\IDE\Extensions\Microsoft
```

from a machine which has Visual Studio installed. Now let's move on to discuss how to customize the publish process.

How to customize publishing In the previous section we saw the visualization of the publish process. The image is shown again to make this easier to follow.
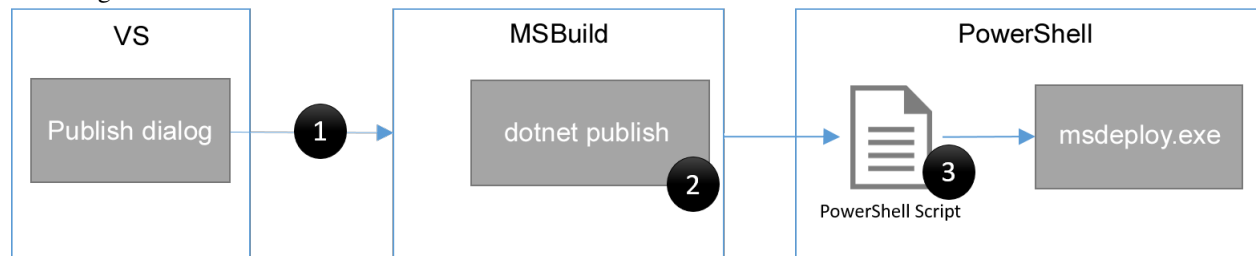
The image above shows the three main extension points, you're most likely to use is #3.

1. Customize the call to `dotnet publish`

Most developers will not need to customize this extension point. Visual Studio starts the publish process by calling an MSBuild target. This target will take care of initializing the environment and calling `dotnet publish` to layout the files. If you need to customize that call in a way that is not enabled by the publish dialog then you can use MSBuild elements in either the project file (.xproj file) or the publish profile (.pubxml file). We won't get into details of how to do that here as it's an advanced scenario that few will need to extend.

2. Customize `dotnet publish`

As stated previously `dotnet publish` is a command line utility that can be used to help publish your ASP.NET Core application. This is a cross platform command line utility (that is, you can use it on Windows, Mac or Linux) and

does not require Visual Studio. If you are working on a team in which some developers are not using Visual Studio, then you may want to script building and publishing. When `dotnet publish` is executed it can be configured to execute custom commands before or after execution. The commands will be listed in project.json in the scripts section.

The supported scripts for publish are prepublish and postpublish. The ASP.NET Core Web Application template uses the prepublish step by default. The relevant snippet from *project.json* is shown below.

```
"scripts": {
  "prepublish": [ "npm install", "bower install", "gulp clean", "gulp min" ]
}
```

Here multiple comma separated calls are declared.

When Visual Studio is used the prepublish and postpublish steps are executed as a part of the call to `dotnet publish`. The postpublish script from *project.json* is executed before the files are published to the remote destination because that takes place immediately after `dotnet publish` completes. In the next step we cover customizing the PowerShell script to control what happens to the files after they reach the target destination.

3. Customize the publish profile PowerShell Script

After creating a publish profile in Visual Studio the PowerShell script `Properties\PublishProfiles\ProfileName.ps1` is created. The script does the following:

1. Runs `dotnet publish`, which will package the web project into a temporary folder to prepare it for the next phase of publishing.

2. The profile PowerShell script is directly invoked. The publish properties and the path to the temporary folder are passed in as parameters. Note, the temporary folder will be deleted on each publish.

As mentioned previously the most important line in the default publish script is the call to `Publish-AspNet`. The call to Publish-AspNet:

- Takes the contents of the folder at $packOutput, which contains the results of `dotnet publish`, and publishes it to the destination.

- The publish properties are passed in the script parameter `$publishProperties`.

- `$publishProperties` is a PowerShell hashtable which contains all the properties declared in the profile .pubxml file. It also includes values for file text replacements or files to exclude. For more info on the values for `$publishProperties` use `Get-Help publish-aspnet -Examples`.

To customize this process, you can edit the PowerShell script directly. To perform an action before publish starts, add the action before the call to `Publish-AspNet`. To have an action performed after publish, add the appropriate calls after Publish-AspNet. When Publish-AspNet is called the contents of the $packOutput directory are published to the destination. For example, if you need add a file to the publish process, just copy it to the correct location in `$packOutput` before `Publish-AspNet` is called. The snippet below shows how to do that.

```
# copy files from image repo to the wwwroot\external-images folder
$externalImagesSourcePath = 'C:\resources\external-images'
$externalImagesDestPath = (Join-Path "$packOutput\wwwroot" 'external-images')
if(-not (Test-Path $externalImagesDestPath)){
  -Item -Path $externalImagesDestPath -ItemType Directory
}

Get-ChildItem $externalImagesSourcePath -File | Copy-Item -Destination $externalImagesDestPath

'Calling Publish-AspNet' | Write-Verbose
# call Publish-AspNet to perform the publish operation
Publish-AspNet -publishProperties $publishProperties -packOutput $packOutput -pubProfilePath $pubPro
```

In this snippet external images are copied from `c:\resources\external-images` to `$packOutput\wwwroot\external-images`. Before starting the copy operation the script ensures that the destination folder exists. Since the copy operation takes place before the call to `Publish-AspNet` the new files will be included in the published content. To perform actions after the files have reached the destination then you can place those commands after the call to `Publish-AspNet`.

You are free to customize, or even completely replace, the Publish-AspNet script provided. As previously mentioned, you will need to preserve the parameter declaration, but the rest is up to you.

1.10.4 Publishing to an Azure Web App with Continuous Deployment

By Erik Reitan

This tutorial shows you how to create an ASP.NET Core web app using Visual Studio and deploy it from Visual Studio to Azure App Service using continuous deployment.

: To complete this tutorial, you need a Microsoft Azure account. If you don't have an account, you can activate your MSDN subscriber benefits or sign up for a free trial.

Sections:

- *Prerequisites*
- *Create an ASP.NET Core web app*
- *Create a web app in the Azure Portal*
- *Enable Git publishing for the new web app*
- *Publish your web app to Azure App Service*
- *Run the app in Azure*
- *Update your web app and republish*
- *View the updated web app in Azure*
- *Additional Resources*

Prerequisites

This tutorial assumes you have already installed the following:

- Visual Studio

- ASP.NET Core (runtime and tooling)

- Git for Windows

Create an ASP.NET Core web app

1. Start Visual Studio.

2. From the **File** menu, select **New > Project**.

3. Select the **ASP.NET Web Application** project template. It appears under **Installed > Templates > Visual C# > Web**. Name the project `SampleWebAppDemo`. Select the **Add to source control** option and click **OK**.

---
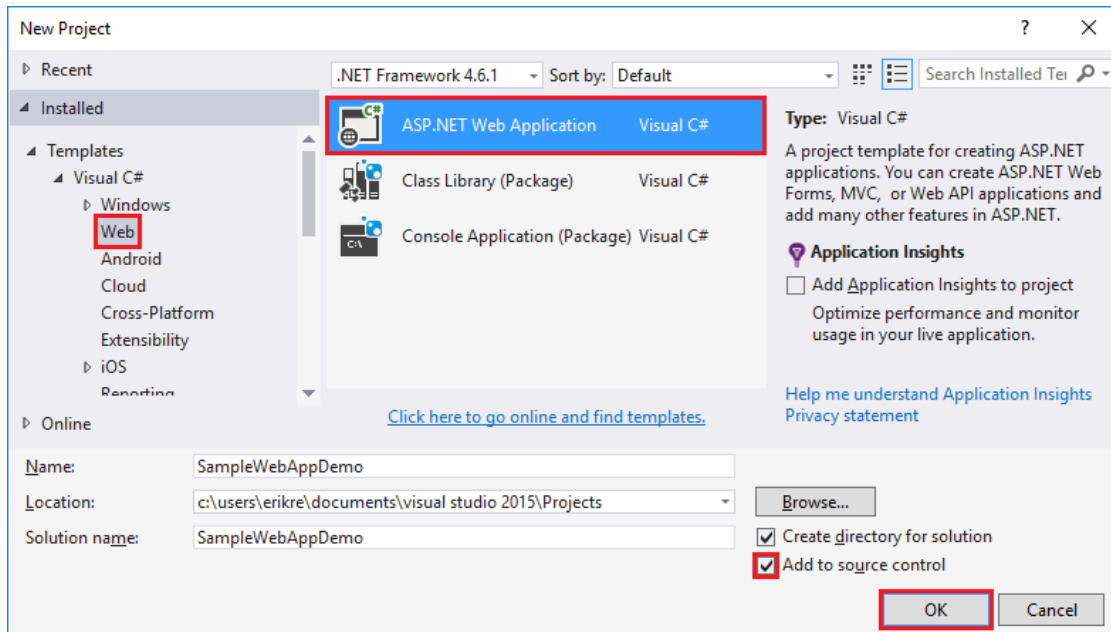
4. In the **New ASP.NET Project** dialog, select the ASP.NET Core **Empty** template, then click **OK**.
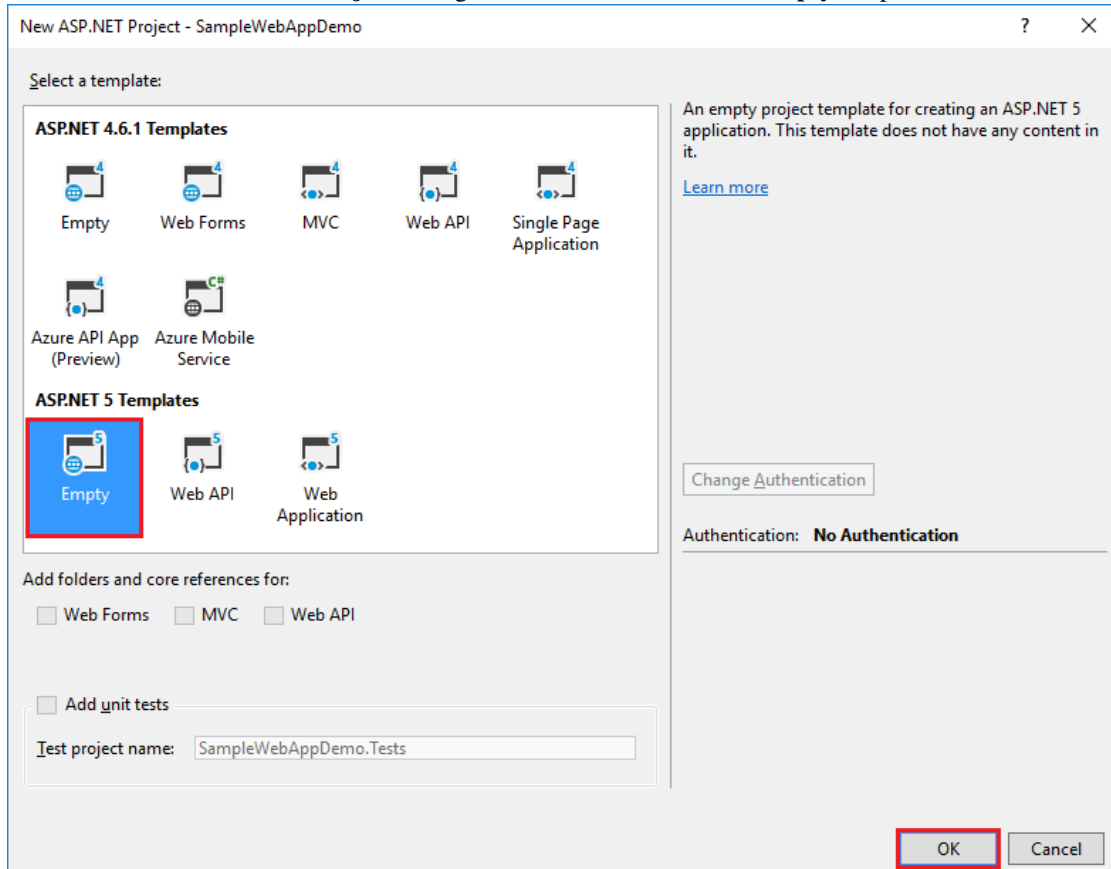
5. From the **Choose Source Control** dialog box, select **Git** as the source control system for the new project.
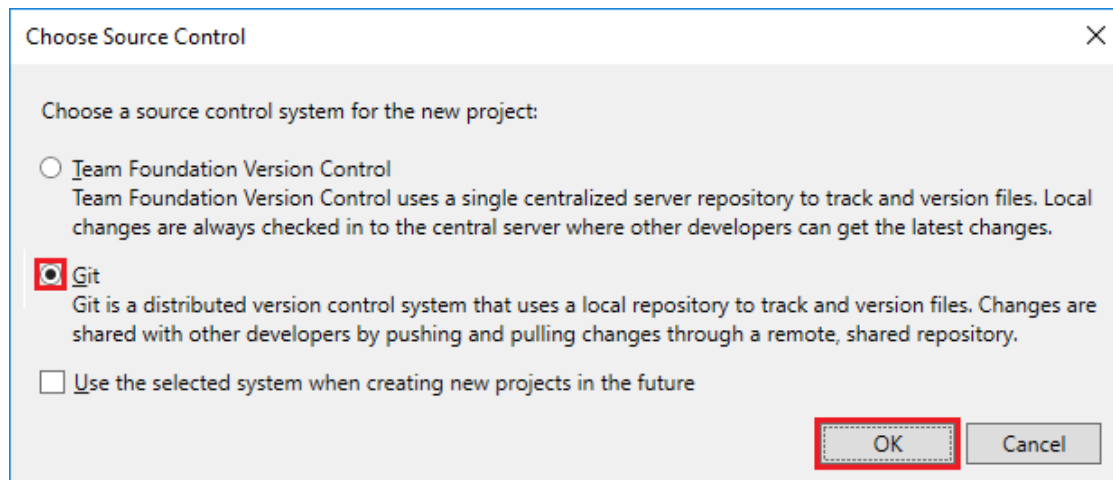
Running the web app locally

1. Once Visual Studio finishes creating the app, run the app by selecting **Debug** -> **Start Debugging**. As an alternative, you can press **F5**.

   It may take time to initialize Visual Studio and the new app. Once it is complete, the browser will show the running app.
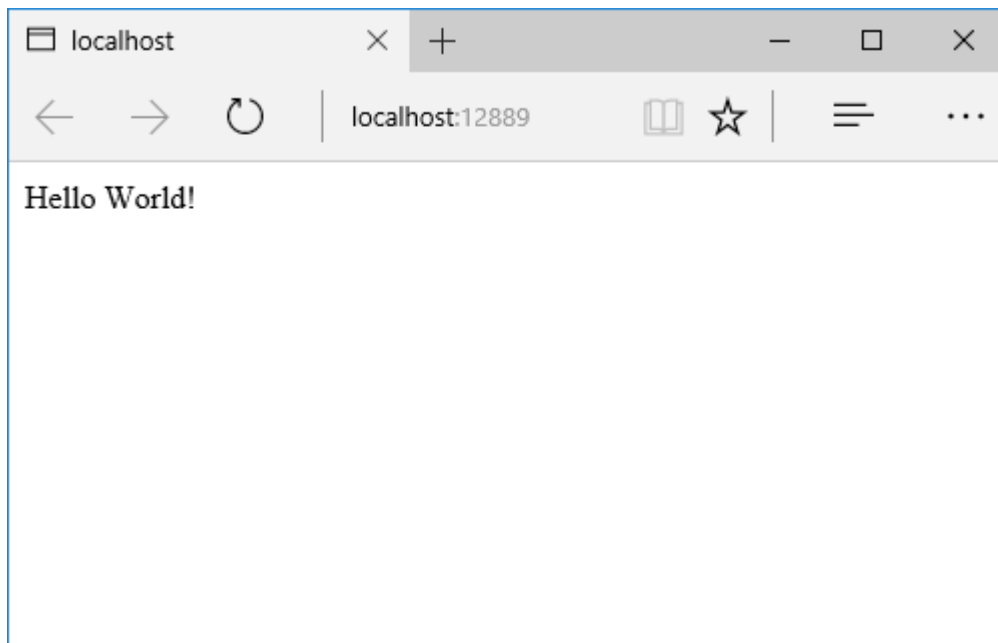
2. After reviewing the running Web app, close the browser and click the "Stop Debugging" icon in the toolbar of Visual Studio to stop the app.
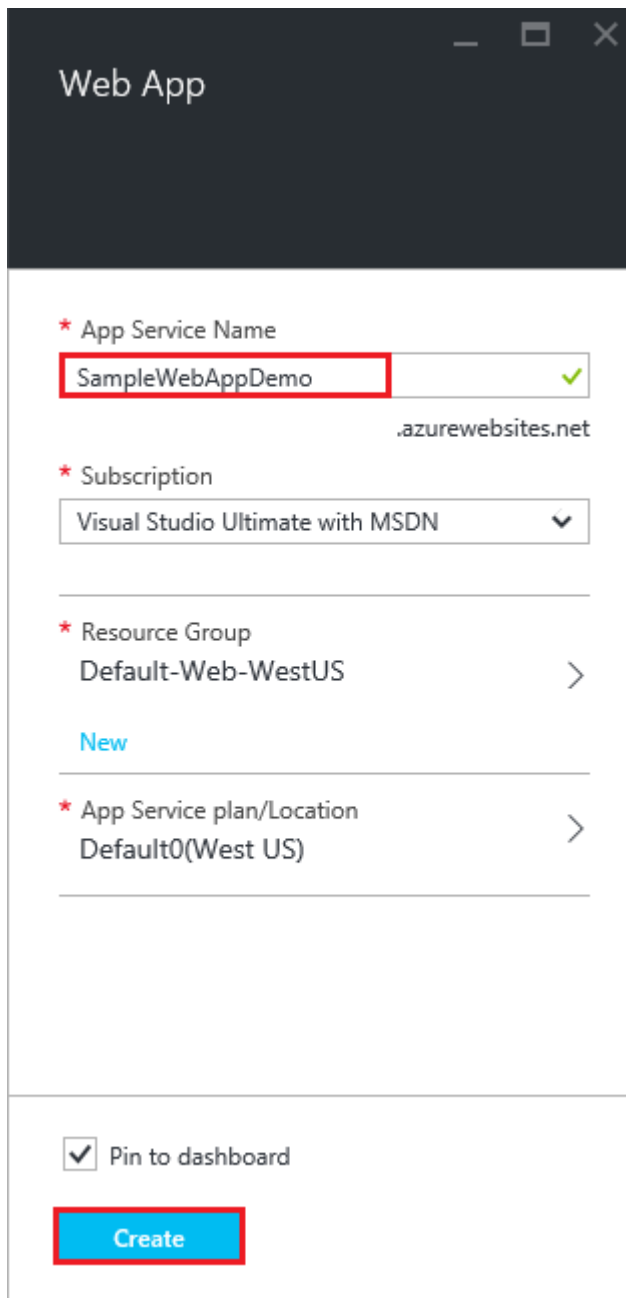
Create a web app in the Azure Portal

The following steps will guide you through creating a web app in the Azure Portal.

1. Log in to the Azure Portal.

2. Click **NEW** at the top left of the Portal.

3. Click **Web + Mobile > Web App**.
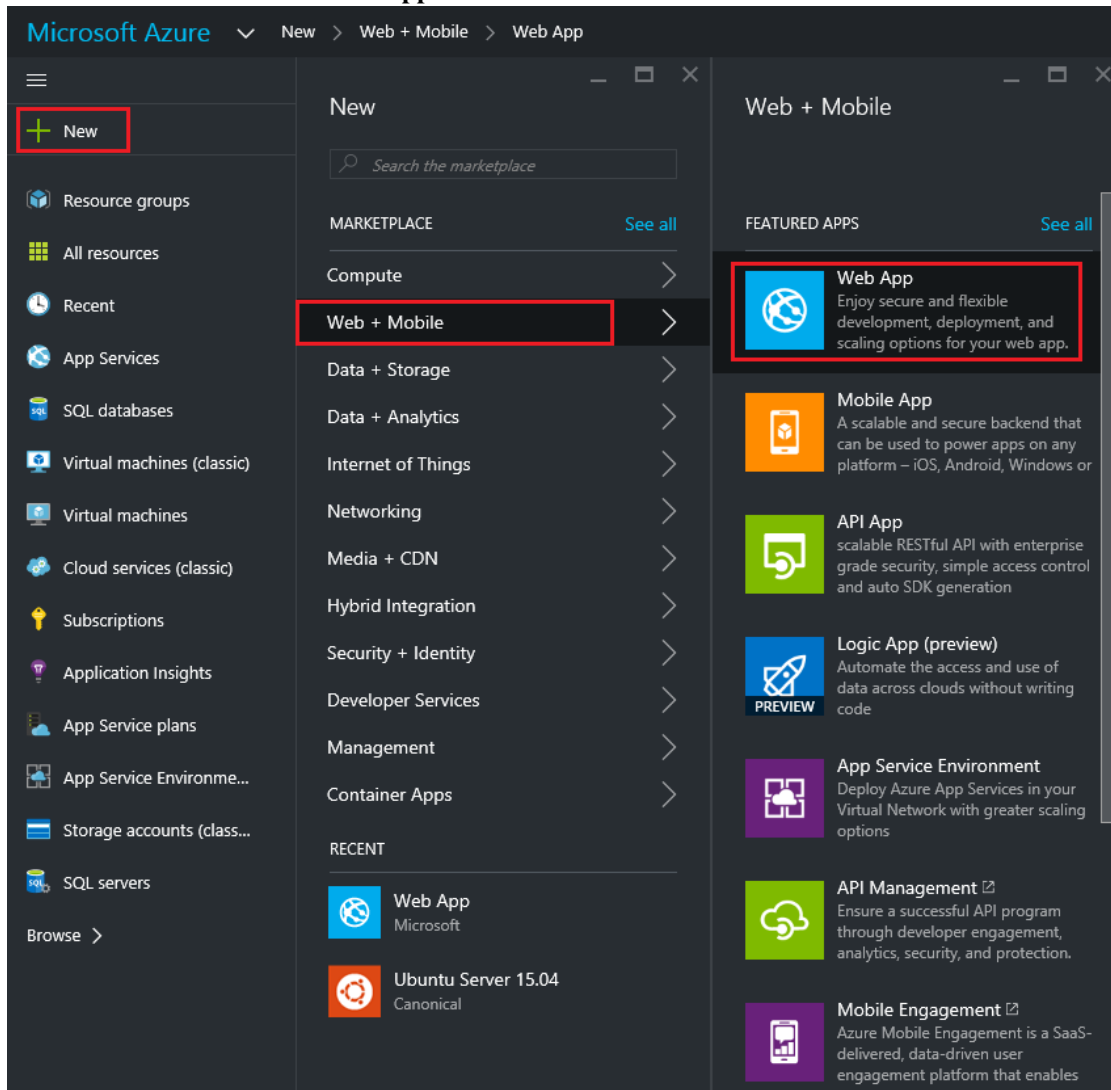
4. In the **Web App** blade, enter a unique value for the **App Service Name**.

: The **App Service Name** name needs to be unique. The portal will enforce this rule when you attempt to enter the name. After you enter a different value, you'll need to substitute that value for each occurrence of **SampleWebAppDemo** that you see in this tutorial.

Also in the **Web App** blade, select an existing **App Service Plan/Location** or create a new one. If you create a new plan, select the pricing tier, location, and other options. For more information on App Service plans, Azure App Service plans in-depth overview.

---

5. Click **Create**. Azure will provision and start your web app.
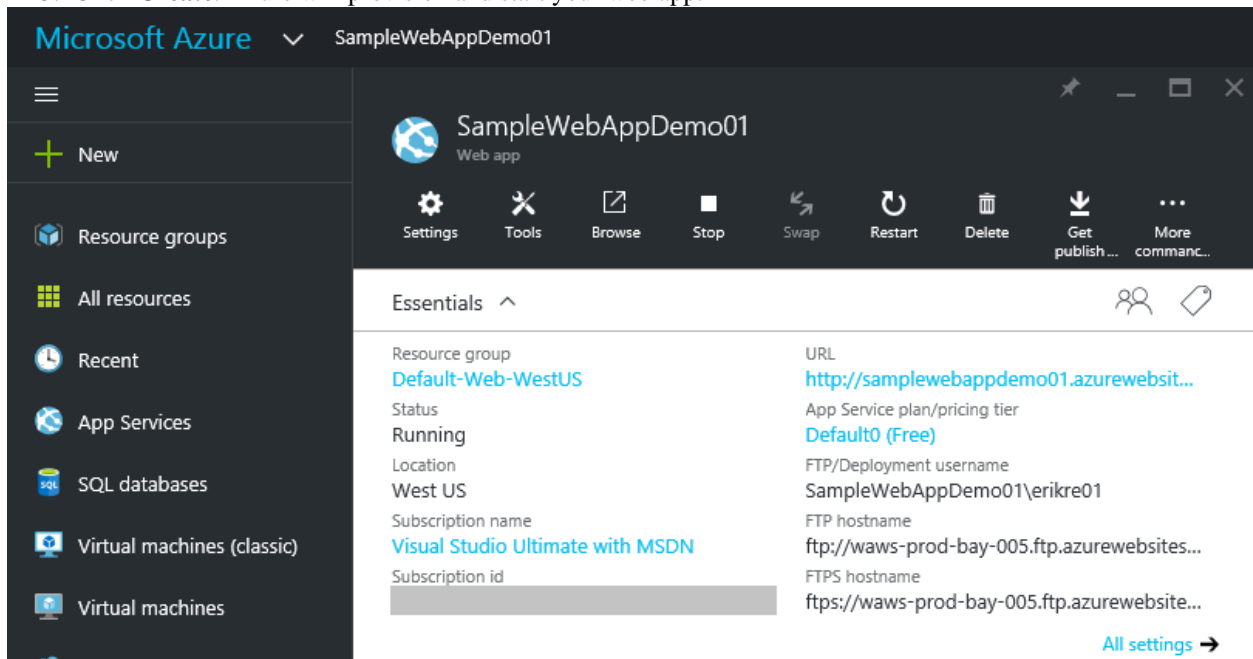
Enable Git publishing for the new web app

Git is a distributed version control system that you can use to deploy your Azure App Service web app. You'll store the code you write for your web app in a local Git repository, and you'll deploy your code to Azure by pushing to a remote repository.

1. Log into the Azure Portal, if you're not already logged in.

2. Click **Browse**, located at the bottom of the navigation pane.

3. Click **Web Apps** to view a list of the web apps associated with your Azure subscription.

4. Select the web app you created in the previous section of this tutorial.

5. If the **Settings** blade is not shown, select **Settings** in the **Web App** blade.

6. In the **Settings** blade, select **Continuous deployment > Choose Source > Local Git Repository**.
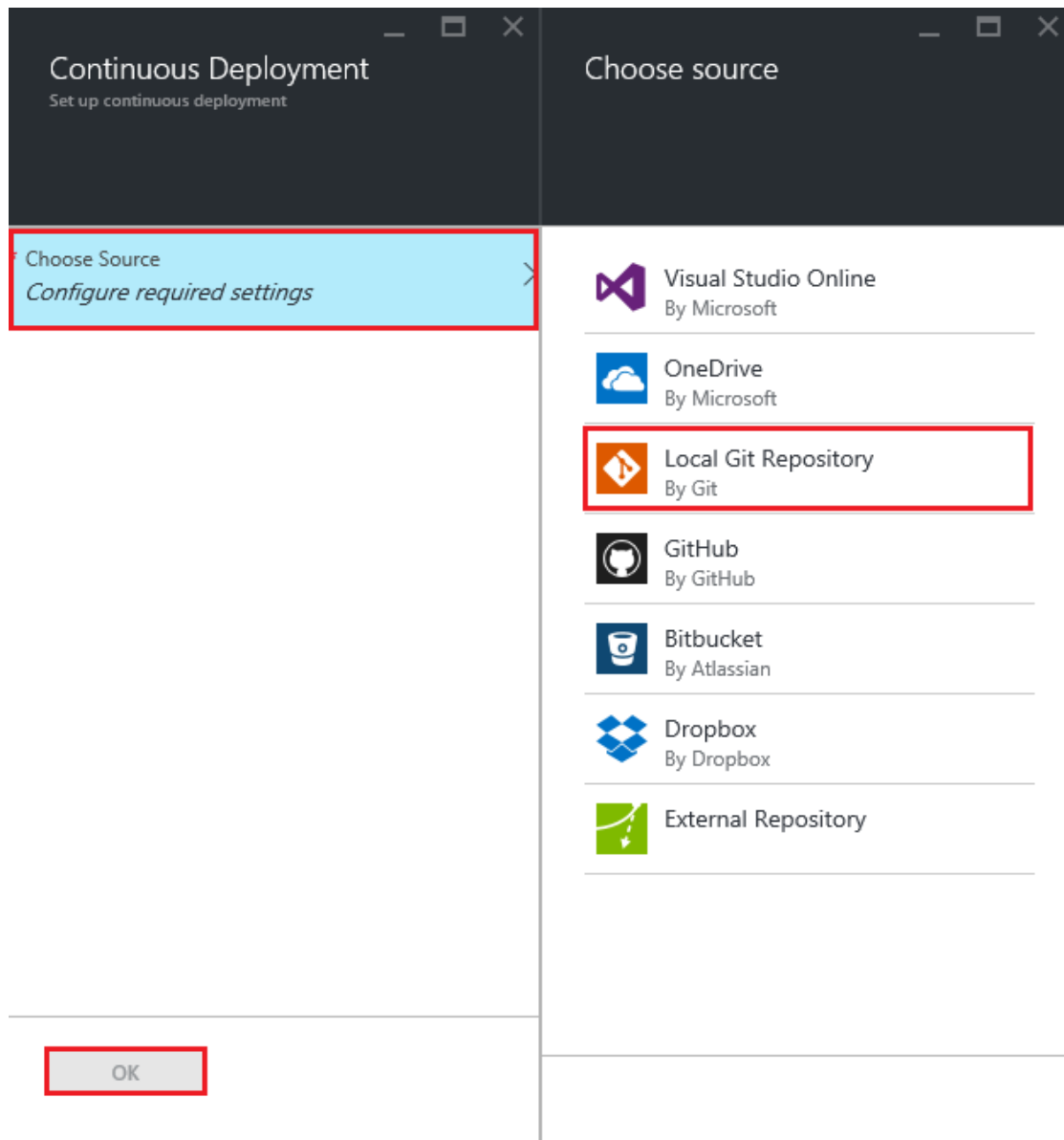
7. Click **OK**.

8. If you have not previously set up deployment credentials for publishing a web app or other App Service app, set them up now:

   - Click **Settings > Deployment credentials**. The **Set deployment credentials** blade will be displayed.

   - Create a user name and password. You'll need this password later when setting up Git.

   - Click **Save**.

9. In the **Web App** blade, click **Settings > Properties**. The URL of the remote Git repository that you'll deploy to is shown under **GIT URL**.

10. Copy the **GIT URL** value for later use in the tutorial.

Publish your web app to Azure App Service

In this section, you will create a local Git repository using Visual Studio and push from that repository to Azure to deploy your web app. The steps involved include the following:

- Add the remote repository setting using your GIT URL value, so you can deploy your local repository to Azure.

- Commit your project changes.

- Push your project changes from your local repository to your remote repository on Azure.

1. In **Solution Explorer** right-click **Solution 'SampleWebAppDemo'** and select **Commit**. The **Team Explorer** will be displayed.
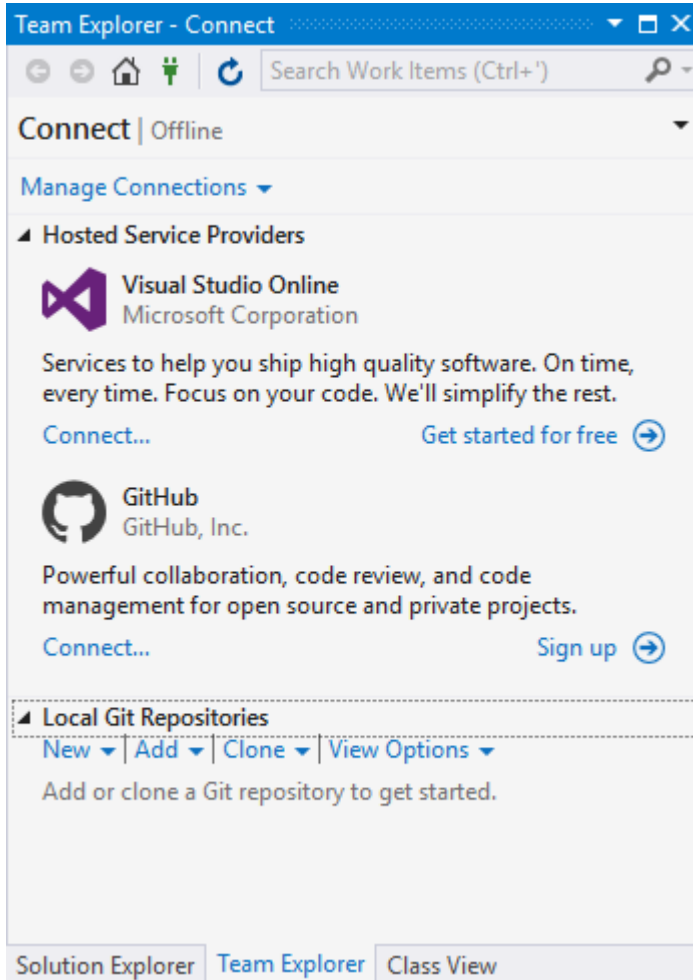
2. In **Team Explorer**, select the **Home** (home icon) **> Settings > Repository Settings**.

3. In the **Remotes** section of the **Repository Settings** select **Add**. The **Add Remote** dialog box will be displayed.

4. Set the **Name** of the remote to **Azure-SampleApp**.

5. Set the value for **Fetch** to the **Git URL** that you copied from Azure earlier in this tutorial. Note that this is the URL that ends with **.git**.
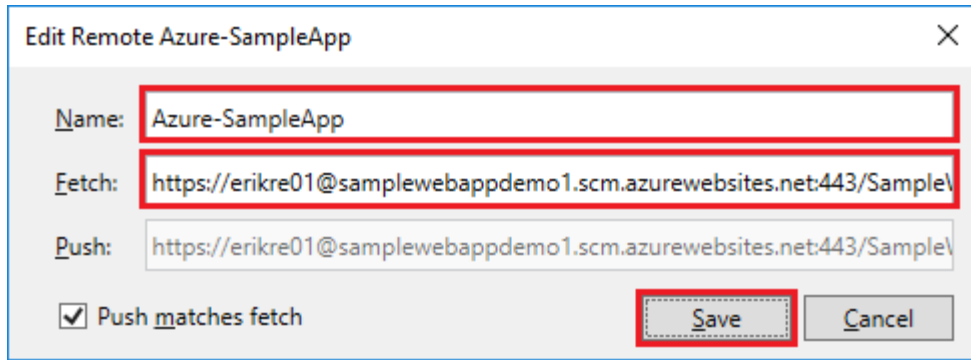
: As an alternative, you can specify the remote repository from the **Command Window** by opening the **Command Window**, changing to your project directory, and entering the command. For example:

```
git remote add Azure-SampleApp https://me@sampleapp.scm.azurewebsites.net:443/SampleApp
```

---

6. Select the **Home** (home icon) **> Settings > Global Settings**. Make sure you have your name and your email address set. You may also need to select **Update**.

7. Select **Home > Changes** to return to the **Changes** view.

8. Enter a commit message, such as **Initial Push #1** and click **Commit**. This action will create a *commit* locally. Next, you need to *sync* with Azure.
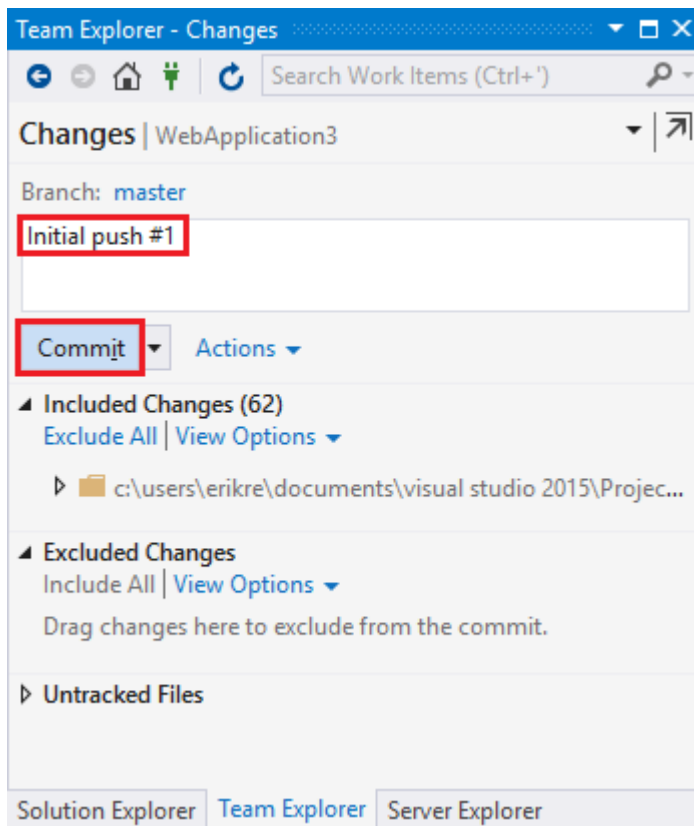
: As an alternative, you can commit your changes from the **Command Window** by opening the **Command Window**, changing to your project directory, and entering the git commands. For example:

```
git add .
git commit -am "Initial Push #1"
```

---

9. Select **Home > Sync > Actions > Open Command Prompt**. The command prompt will open to your project directory.

10. Enter the following command in the command window:

```
git push -u Azure-SampleApp master
```

11. Enter your Azure **deployment credentials** password that you created earlier in Azure.

: Your password will not be visible as you enter it.

This command will start the process of pushing your local project files to Azure. The output from the above command ends with a message that deployment was successful.

remote: Finished successfully.

---

remote: Running post deployment command(s)...

remote: Deployment successful.

To https://username@samplewebappdemo01.scm.azurewebsites.net:443/SampleWebAppDemo01.git

* [new branch] master -> master

Branch master set up to track remote branch master from Azure-SampleApp.

: If you need to collaborate on a project, you should consider pushing to GitHub in between pushing to Azure.

Verify the Active Deployment

You can verify that you successfully transferred the web app from your local environment to Azure. You'll see the listed successful deployment.

1. In the Azure Portal, select your web app. Then, select **Settings > Continuous deployment**.

Run the app in Azure

Now that you have deployed your web app to Azure, you can run the app.

This can be done in two ways:

- In the Azure Portal, locate the web app blade for your web app, and click **Browse** to view your app in your default browser.

- Open a browser and enter the URL for your web app. For example:

  ```
  http://SampleWebAppDemo.azurewebsites.net
  ```

Update your web app and republish

After you make changes to your local code, you can republish.

1. In **Solution Explorer** of Visual Studio, open the *Startup.cs* file.

2. In the `Configure` method, modify the `Response.WriteAsync` method so that it appears as follows:

```
await context.Response.WriteAsync("Hello World! Deploy to Azure.");
```

3. Save changes to *Startup.cs*.

4. In **Solution Explorer**, right-click **Solution 'SampleWebAppDemo'** and select **Commit**. The **Team Explorer** will be displayed.

5. Enter a commit message, such as:

```
Update #2
```

6. Press the **Commit** button to commit the project changes.

7. Select **Home > Sync > Actions > Push**.

   : As an alternative, you can push your changes from the **Command Window** by opening the **Command Window**, changing to your project directory, and entering a git command. For example:

   ```
   git push -u Azure-SampleApp master
   ```

View the updated web app in Azure

View your updated web app by selecting **Browse** from the web app blade in the Azure Portal or by opening a browser and entering the URL for your web app. For example:

```
http://SampleWebAppDemo.azurewebsites.net
```

Additional Resources

- *Publishing and Deployment*
- Project Kudu

1.10.5 Publishing to a Windows Virtual Machine on Azure

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.10.6 Publish to a Linux Production Environment

By Sourabh Shirhatti

In this guide, we will cover setting up a production-ready ASP.NET environment on an Ubuntu 14.04 Server.

We will take an existing ASP.NET Core application and place it behind a reverse-proxy server. We will then setup the reverse-proxy server to forward requests to our Kestrel web server.

Additionally we will ensure our web application runs on startup as a daemon and configure a process management tool to help restart our web application in the event of a crash to guarantee high availability.

Sections:

- *Prerequisites*
- *Copy over your app*
- *Configure a reverse proxy server*
- *Monitoring our Web Application*
- *Start our web application on startup*
- *Recovering from an ungraceful shutdown*
- *Viewing logs*

Prerequisites

1. Access to an Ubuntu 14.04 Server with a standard user account with sudo privilege.

2. An existing ASP.NET Core application.

Copy over your app

Run `dotnet publish` from your dev environment to package your application into a self-contained directory that can run on your server.

Before we proceed, copy your ASP.NET Core application to your server using whatever tool (SCP, FTP, etc) integrates into your workflow. Try and run the app and navigate to `http://<serveraddress>:<port>` in your browser to see if the application runs fine on Linux. I recommend you have a working app before proceeding.

: You can use *Yeoman* to create a new ASP.NET Core application for a new project.

Configure a reverse proxy server

A reverse proxy is a common setup for serving dynamic web applications. The reverse proxy terminates the HTTP request and forwards it to the ASP.NET application.

Why use a reverse-proxy server?

Kestrel is great for serving dynamic content from ASP.NET, however the web serving parts aren't as feature rich as full-featured servers like IIS, Apache or Nginx. A reverse proxy-server can allow you to offload work like serving static content, caching requests, compressing requests, and SSL termination from the HTTP server. The reverse proxy server may reside on a dedicated machine or may be deployed alongside an HTTP server.

For the purposes of this guide, we are going to use a single instance of Nginx that runs on the same server alongside your HTTP server. However, based on your requirements you may choose a different setup.

Install Nginx

```
sudo apt-get install nginx
```

: If you plan to install optional Nginx modules you may be required to build Nginx from source.

We are going to `apt-get` to install Nginx. The installer also creates a System V init script that runs Nginx as daemon on system startup. Since we just installed Nginx for the first time, we can explicitly start it by running

```
sudo service nginx start
```

At this point you should be able to navigate to your browser and see the default landing page for Nginx.

Configure Nginx

We will now configure Nginx as a reverse proxy to forward requests to our ASP.NET application

We will be modifying the `/etc/nginx/sites-available/default`, so open it up in your favorite text editor and replace the contents with the following.

```
server {
    listen 80;
    location / {
        proxy_pass http://unix:/var/aspnet/HelloMVC/kestrel.sock;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection keep-alive;
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

This is one of the simplest configuration files for Nginx that forwards incoming public traffic on your port `80` to a unix socket that your web application will listen on. You can specify this Unix socket in your *project.json* file.

1.6: project.json

```
"commands": {
    "web": "Microsoft.AspNetCore.Server.Kestrel --server.urls http://unix:/var/aspnet/HelloMVC/kestre
},
```

You might want to look at `/etc/nginx/nginx.conf` to configure your nginx environment.

Once you have completed making changes to your nginx configuration you can run `sudo nginx -t` to verify the syntax of your configuration files. If the configuration file test is successful you can ask nginx to pick up the changes by running `sudo nginx -s reload`.

Monitoring our Web Application

Nginx will forward requests to your Kestrel server, however unlike IIS on Windows, it does not mangage your Kestrel process. In this tutorial, we will use supervisor to start our application on system boot and restart our process in the event of a failure.

Installing supervisor

```
sudo apt-get install supervisor
```

: `supervisor` is a python based tool and you can acquire it through pip or easy_install instead.

Configuring supervisor

Supervisor works by creating child processes based on data in its configuration file. When a child process dies, supervisor is notified via the `SIGCHILD` signal and supervisor can react accordingly and restart your web application.

To have supervisor monitor our application, we will add a file to the `/etc/supervisor/conf.d/` directory.

1.7: /etc/supervisor/conf.d/hellomvc.conf

```
[program:hellomvc]
command=bash /var/aspnet/HelloMVC/approot/web
autostart=true
autorestart=true
stderr_logfile=/var/log/hellomvc.err.log
stdout_logfile=/var/log/hellomvc.out.log
environment=Hosting__Environment=Production
user=www-data
stopsignal=INT
```

Once you are done editing the configuration file, restart the `supervisord` process to change the set of programs controlled by supervisord.

```
sudo service supervisor stop
sudo service supervisor start
```

Start our web application on startup

In our case, since we are using supervisor to manage our application, the application will be automatically started by supervisor. Supervisor uses a System V Init script to run as a daemon on system boot and will susbsequently launch your application. If you chose not to use supervisor or an equivalent tool, you will need to write a `systemd` or `upstart` or `SysVinit` script to start your application on startup.

Recovering from an ungraceful shutdown

If your web application is terminated with a `SIGKILL` signal or the if host experiences a loss of power, `Kestrel` will not shut down gracefully and remove the socket file. To prevent subsequents attempts to restart your application from failing due to `EADDRINUSE address already in use`, you can modify the shell script used to bootstrap your application to remove the socket file if present.

1.8: /var/aspnet/HelloMVC/approot/web

```
if [ -f "/var/aspnet/HelloMVC/kestrel.sock" ]; then
  rm "/var/aspnet/HelloMVC/kestrel.sock"
fi
```

Viewing logs

Supervisord logs messages about its own health and its subprocess' state changes to the activity log. The path to the activity log is configured via the `logfile` parameter in the configuration file.

```
sudo tail -f /var/log/supervisor/supervisord.log
```

You can redirect application logs (`STDOUT` and `STERR`) in the program section of your configuration file.

```
tail -f /var/log/hellomvc.out.log
```

1.10.7 How to Customize Publishing

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.11 Guidance for Hosting Providers

1.11.1 ASP.NET Core Module Configuration Reference

By Luke Latham, Rick Anderson and Sourabh Shirhatti

In ASP.NET Core, the web application is hosted by an external process outside of IIS. The ASP.NET Core Module is an IIS 7.5+ module which is responsible for process management of ASP.NET Core http listeners and to proxy requests to processes that it manages. This document provides an overview of how to configure the ASP.NET Core Module for shared hosting of ASP.NET Core.

Sections:

- *Installing the ASP.NET Core Module*
- *Configuring the ASP.NET Core Module*
    - *Configuration Attributes*
    - *Child Elements*
- *ASP.NET Core Module* app_offline.htm
- *ASP.NET Core Module Start Up Error Page*
- *ASP.NET Core Module configuration examples*
    - *Log creation and redirection*
    - *Setting environment variables*

Installing the ASP.NET Core Module

Install the .NET Core Windows Server Hosting bundle on the server. The bundle will install the .NET Core Runtime, .NET Core Library, and the ASP.NET Core Module.

Configuring the ASP.NET Core Module

The ASP.NET Core Module is configured via a site or application *web.config* file and has its own configuration section within `system.webServer - aspNetCore`.

Configuration Attributes

Attribute	Description
processPath	Required string attribute. Path to the executable or script that will launch a process listening for HTTP requests. Relative paths are supported. If the path begins with '.' the path is considered to be relative to the site root. There is no default value.
arguments	Optional string attribute. Arguments to the executable or script specified in **processPath**. The default value is an empty string.
startupTimeLimit	Optional integer attribute. Duration in seconds for which the the handler will wait for the executable or script to start a process listening on the port. If this time limit is exceeded, the handler will kill the process and attempt to launch it again **startupRetryCount** times. The default value is 120.
shutdownTimeLimit	Optional integer attribute. Duration in seconds for which the the handler will wait for the executable or script to gracefully shutdown when the *app_offline.htm* file is detected The default value is 10.
rapidFailsPerMinute	Optional integer attribute. Specifies the number of times the process specified in **processPath** is allowed to crash per minute. If this limit is exceeded, the handler will stop launching the process for the remainder of the minute.

Child Elements

Attribute	Description
environmentVariables	Configures **environmentVariables** collection for the process specified in **processPath**.
recycleOnFileChange	Specify a list of files to monitor. If any of these files get updated/deleted, the Core Module will restart the backend process.

ASP.NET Core Module *app_offline.htm*

If you place a file with the name *app_offline.htm* at the root of a web application directory, the ASP.NET Core Module will attempt to gracefully shut-down the application and stop processing any new incoming requests. If the application is still running after `shutdownTimeLimit` number of seconds, the ASP.NET Core Module will kill the running process.

While the *app_offline..htm* file is present, the ASP.NET Core Module will repond to all requests by sending back the contents of the *app_offline.htm* file. Once the *app_offline.htm* file is removed, the next request loads the application, which then responds to requests.

ASP.NET Core Module Start Up Error Page
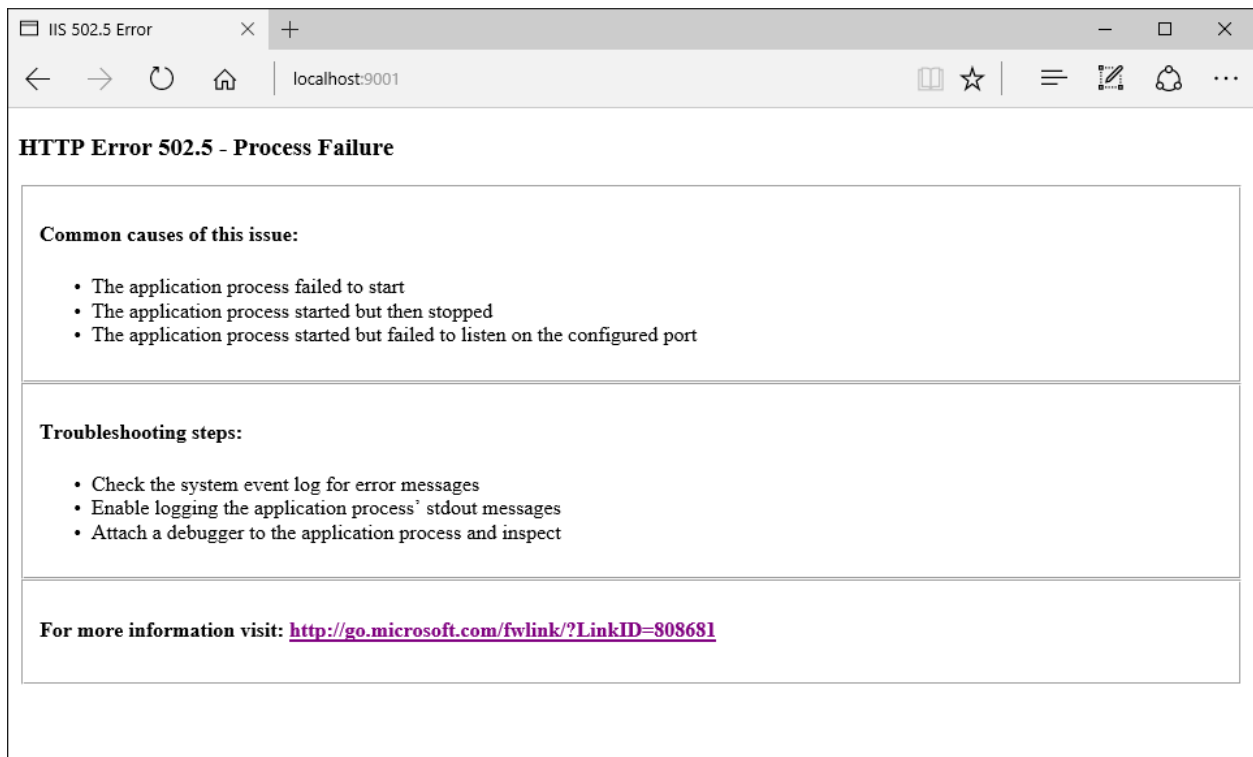
If the ASP.NET Core Module fails to launch the backend process or the backend process starts but fails to listen on the configured port, you will see an HTTP 502.5 status code page. To supress this page and revert to the default IIS 502 status code page, use the `disableStartUpErrorPage` attribute. Look at the HTTP Errors attribute to override this page with a custom error page.

ASP.NET Core Module configuration examples

Log creation and redirection

For logs to be saved, you must create the log directory. The ASP.NET Core Module can redirect `stdout` and `stderr` logs to disk by setting the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element. Logs are not rotated. It is the responsibilty of the hoster to limit the disk space the logs consume.

```xml
<aspNetCore processPath="dotnet"
        arguments=".\MyApp.dll"
        stdoutLogEnabled="true"
        stdoutLogFile=".\logs\stdout">
</aspNetCore>
```

Setting environment variables

The ASP.NET Core Module allows you specify environment variables for the process specified in the `processPath` setting by specifying them in `environmentVariables` child attribute to the `aspNetCore` attribute.

```xml
<aspNetCore processPath="dotnet"
        arguments=".\MyApp.dll"
        stdoutLogEnabled="true"
        stdoutLogFile=".\logs\stdout">
  <environmentVariables>
    <environmentVariable name="DEMO" value="demo_value" />
  </environmentVariables>
</aspNetCore>
```

1.11.2 Directory Structure

By Luke Latham

In ASP.NET Core, the application directory, *publish*, is comprised of application files, config files, static assets, packages, and the runtime (for self-contained apps). This is the same directory structure as previous versions of ASP.NET, where the entire application lives inside the web root directory.

App Type	Directory Structure

Portable	• publish*
	
	    – logs* (if included in publishOptions)
	
	    – refs*
	
	    – runtimes*
	
	    – Views* (if included in publishOptions)
	
	    – wwwroot* (if included in publishOptions)
	
	    – .dll files
	
	    – myapp.deps.json
	
	    – myapp.dll
	
	    – myapp.pdb
	
	    – myapp.runtimeconfig.json
	
	    – web.config (if included in publishOptions)

Self-contained	• publish*
	
	    – logs* (if included in publishOptions)
	
	    – refs*
	
	    – Views* (if included in publishOptions)
	
	    – wwwroot* (if included in publishOptions)
	
	    – .dll files
	
	    – myapp.deps.json
	
	    – myapp.exe
	
	    – myapp.pdb
	
	    – myapp.runtimeconfig.json
	
	    – web.config (if included in publishOptions)

* Indicates a directory

The contents of the *publish* directory represent the *content root path*, also called the *application base path*, of the deployment. Whatever name is given to the *publish* directory in the deployment, its location serves as the server's physical path to the hosted application. The *wwwroot* directory, if present, only contains static assets. The *logs* directory may be included in the deployment by creating it in the project and adding it to **publishOptions** of *project.json* or by physically creating the directory on the server.

The deployment directory requires Read/Execute permissions, while the *logs* directory requires Read/Write permissions. Additional directories where assets will be written require Read/Write permissions.

1.11.3 Application Pools

By Sourabh Shirhatti

When hosting multiple web sites on a single server, you should consider isolating the applications from each other by

---

running each application in its own application pool. This document provides an overview of how to set up Application Pools to securely host multiple web sites on a single server.

Application Pool Identity Account

An application pool identity account allows you to run an application under a unique account without having to create and manage domains or local accounts. On IIS 8.0+ the IIS Admin Worker Process (WAS) will create a virtual account with the name of the new application pool and run the application pool's worker processes under this account by default.

Configuring IIS Application Pool Identities

In the IIS Management Console, under **Advanced Settings** for your application pool ensure that *Identity* list item is set to use **ApplicationPoolIdentity** as shown in the image below.
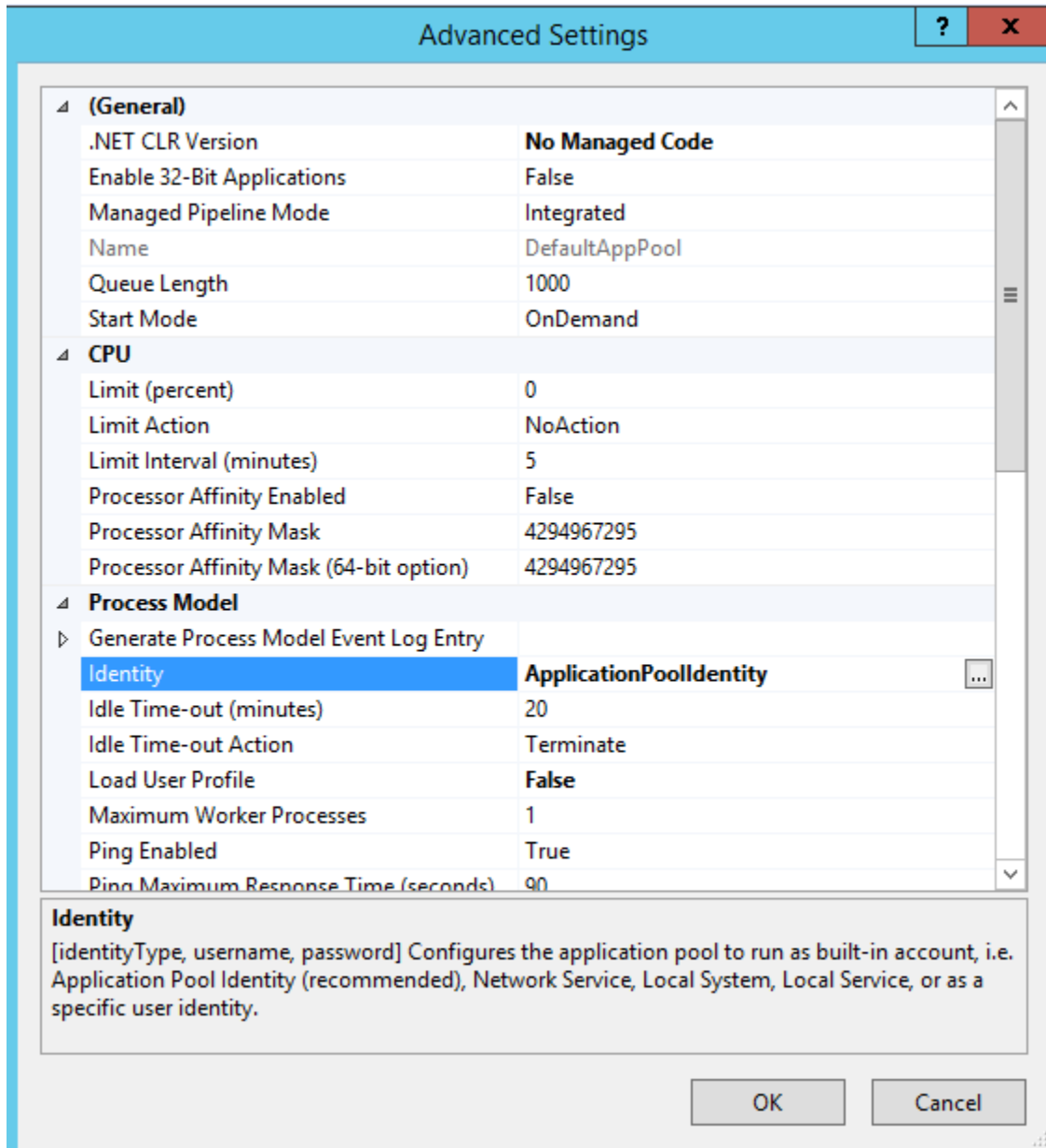
Securing Resources

The IIS management process creates a secure identifier with the name of the application pool in the Windows Security System. Resources can be secured by using this identity, however this identity is not a real user account and will not show up in the Windows User Management Console.

To grant the IIS worker process access to your application, you will need to modify the Access Control List (ACL) for the the directory containing your application.

1. Open Windows Explorer and navigate to the directory.

2. Right click on the directory and click properties.

3. Under the **Security** tab, click the **Edit** button and then the **Add** button

4. Click the **Locations** and make sure you select your server.
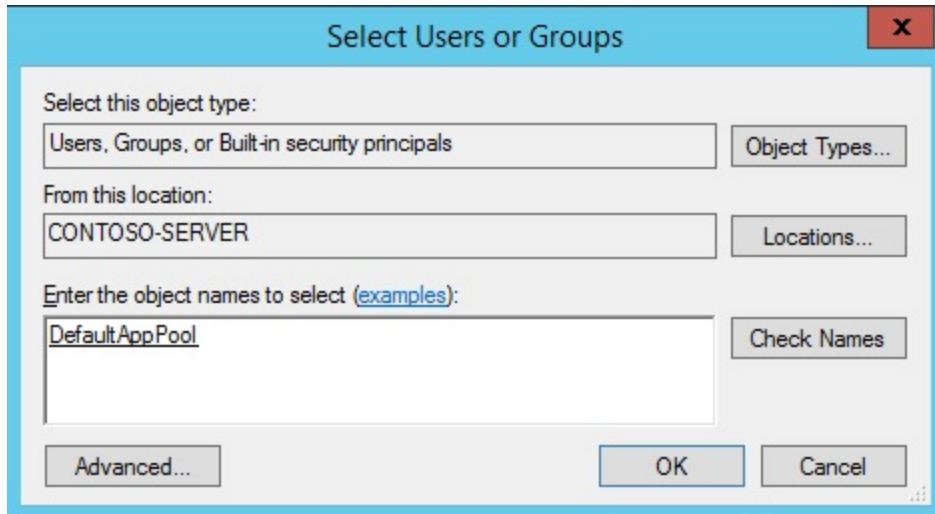
5. Enter **IIS AppPool\DefaultAppPool** in **Enter the object names to select** textbox.

6. Click the **Check Names** button and then click **OK**.

You can also do this via the command-line by using **ICACLS** tool.

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool" :F
```

1.11.4 Servicing

By Sourabh Shirhatti, Daniel Roth

.NET Core supports servicing of runtime components and packages to patch any vulnerabilities when they are discovered. For information on how to enable servicing for applications in a hosted environment please refer to the .NET Core Servicing documentation.

1.11.5 Data Protection

By Sourabh Shirhatti

The ASP.NET Core data protection stack provides a simple and easy to use cryptographic API a developer can use to protect data, including key management and rotation. This document provides an overview of how to configure Data Protection on your server to enable developers to use data protection.

Configuring Data Protection

> **:** Data Protection is used by various ASP.NET middleware, including those used in authentication. The default behavior on IIS hosted web sites is to store keys in memory and discard them when the process restarts. This behavior will have side effects, for example, discarding keys invalidate any cookies written by the cookie authentication and users will have to login again.

To automatically persist keys for an application hosted in IIS, you must create registry hives for each application pool. Use the Provisioning PowerShell script for each application pool you will be hosting ASP.NET Core applications under. This script will create a special registry key in the HKLM registry that is ACLed only to the worker process account. Keys are encrypted at rest using DPAPI.

: A developer can configure their applications Data Protection APIs to store data on the file system. Data Protection can be configured by the developer to use a UNC share to store keys, to enable load balancing. A hoster should ensure that the file permissions for such a share are limited to the Windows account the application runs as. In addition a developer may choose to protect keys at rest using an X509 certificate. A hoster may wish to consider a mechanism to allow users to upload certificates, place them into the user's trusted certificate store and ensure they are available on all machines the users application will run on.

Machine Wide Policy

The data protection system has limited support for setting default *machine-wide policy* for all applications that consume the data protection APIs. See the *data protection* documentation for more details.

1.12 Security

1.12.1 Authentication

Introduction to Identity

By Pranav Rastogi, Rick Anderson, Tom Dykstra, Jon Galloway and Erik Reitan

ASP.NET Core Identity is a membership system which allows you to add login functionality to your application. Users can create an account and login with a user name and password or they can use an external login providers such as Facebook, Google, Microsoft Account, Twitter and more.

You can configure ASP.NET Core Identity to use a SQL Server database to store user names, passwords, and profile data. Alternatively, you can use your own persistent store to store data in another other persistent storage, such as Azure Table Storage.

Overview of Identity

In this topic, you'll learn how to use ASP.NET Core Identity to add functionality to register, log in, and log out a user. You can follow along step by step or just read the details. For more detailed instructions about creating apps using ASP.NET Core Identity, see the Next Steps section at the end of this article.

1. Create an ASP.NET Core Web Application project in Visual Studio with Individual User Accounts.

   In Visual Studio, select **File** -> **New** -> **Project**. Then, select the **ASP.NET Web Application** from the **New Project** dialog box. Continue by selecting an ASP.NET Core **Web Application** with **Individual User Accounts** as the authentication method.
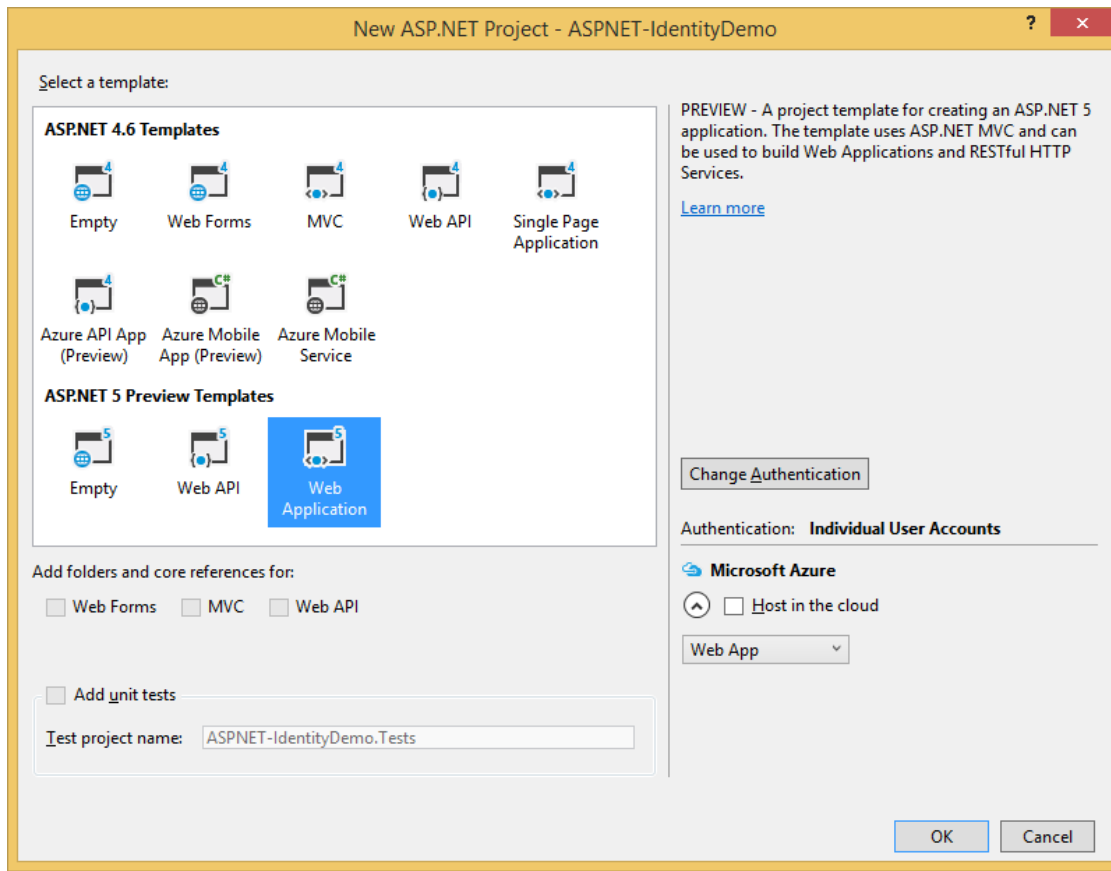
The created project contains the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` package, which will persist the identity data and schema to SQL Server using Entity Framework Core.

: In Visual Studio, you can view NuGet packages details by selecting **Tools** -> **NuGet Package Manager** -> **Manage NuGet Packages for Solution**. You also see a list of packages in the dependencies section of the *project.json* file within your project.

The identity services are added to the application in the `ConfigureServices` method in the `Startup` class:

```csharp
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddEntityFramework()
        .AddSqlServer()
        .AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(Configuration["Data:DefaultConnection:ConnectionString"]));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
```

```
        services.AddTransient<IEmailSender, AuthMessageSender>();
        services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

These services are then made available to the application through *dependency injection*.

Identity is enabled for the application by calling `UseIdentity` in the `Configure` method of the `Startup` class. This adds cookie-based authentication to the request pipeline.

```csharp
// This method gets called by the runtime. Use this method to configure the HTTP request pipelin
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFac
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseBrowserLink();
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseIISPlatformHandler(options => options.AuthenticationDescriptions.Clear());

    app.UseStaticFiles();

    app.UseIdentity();

    // To configure external authentication please see http://go.microsoft.com/fwlink/?LinkID=53

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```
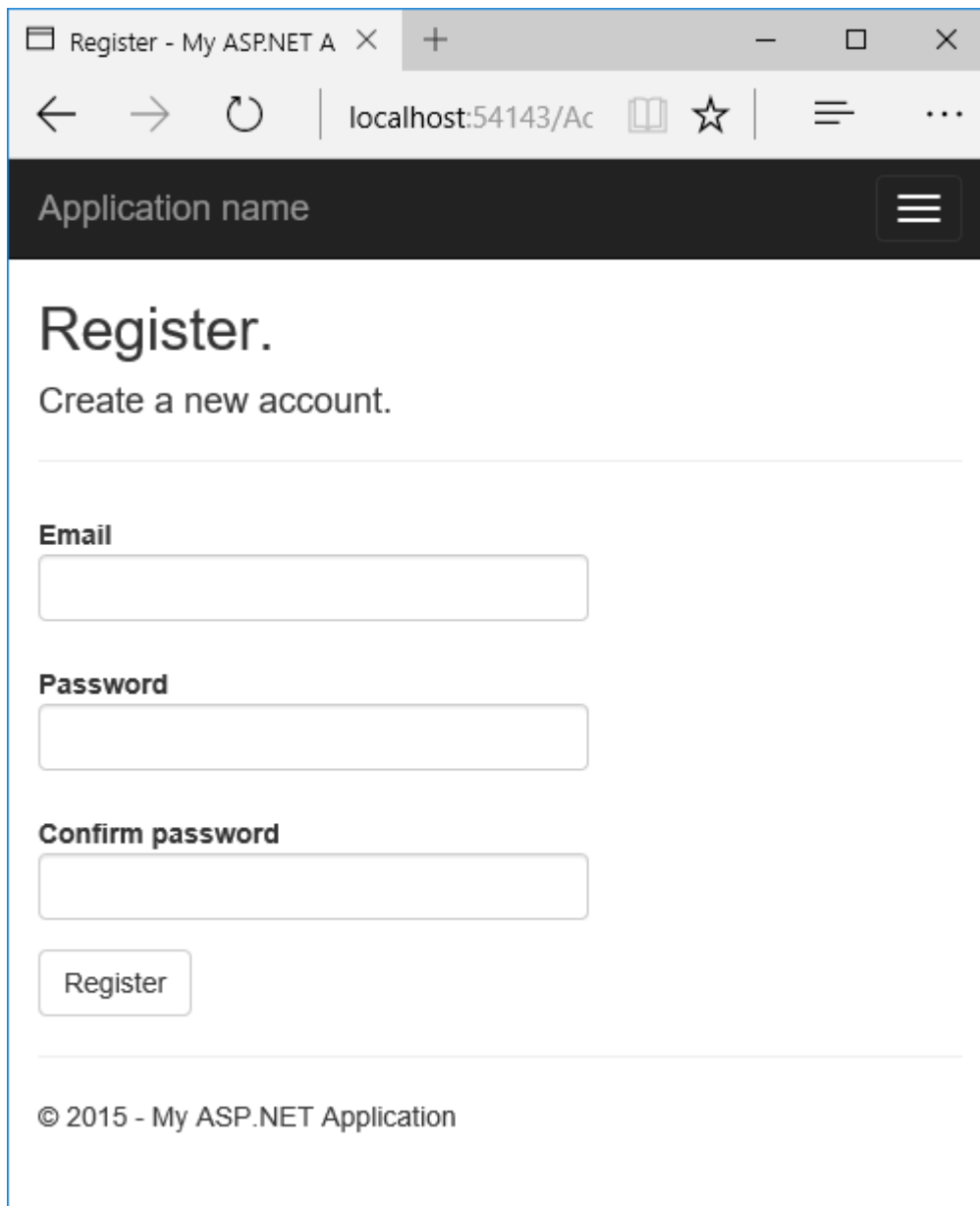
For more information about the application start up process, see *Application Startup*.

2. Creating a user.

   Launch the application from Visual Studio (**Debug** -> **Start Debugging**) and then click on the **Register** link in the browser to create a user. The following image shows the Register page which collects the user name and password.

When the user clicks the **Register** link, the `UserManager` and `SignInManager` services are injected into the Controller:

```
public class AccountController : Controller
{
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly SignInManager<ApplicationUser> _signInManager;
    private readonly IEmailSender _emailSender;
    private readonly ISmsSender _smsSender;
    private static bool _databaseChecked;
    private readonly ILogger _logger;

    public AccountController(
        UserManager<ApplicationUser> userManager,
        SignInManager<ApplicationUser> signInManager,
```

```
                IEmailSender emailSender,
                ISmsSender smsSender,
                ILoggerFactory loggerFactory)
        {
            _userManager = userManager;
            _signInManager = signInManager;
            _emailSender = emailSender;
            _smsSender = smsSender;
            _logger = loggerFactory.CreateLogger<AccountController>();
        }

        //
        // GET: /Account/Login
```

Then, the **Register** action creates the user by calling `CreateAsync` function of the `UserManager` object, as shown below:

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // For more information on how to enable account confirmation and password reset ple
            // Send an email with this link
            //var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            //var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, co
            //await _emailSender.SendEmailAsync(model.Email, "Confirm your account",
            //    "Please confirm your account by clicking this link: <a href=\"" + callbackUrl
            await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation(3, "User created a new account with password.");
            return RedirectToAction(nameof(HomeController.Index), "Home");
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

3. Log in.

   If the user was successfully created, the user is logged in by the `SignInAsync` method, also contained in the `Register` action. By signing in, the `SignInAsync` method stores a cookie with the user's claims.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
```

```
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // For more information on how to enable account confirmation and password reset ple
            // Send an email with this link
            //var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            //var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, co
            //await _emailSender.SendEmailAsync(model.Email, "Confirm your account",
            //    "Please confirm your account by clicking this link: <a href=\"" + callbackUrl
            await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation(3, "User created a new account with password.");
            return RedirectToAction(nameof(HomeController.Index), "Home");
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

The above `SignInAsync` method calls the below `SignInAsync` task, which is contained in the `SignInManager` class.

If needed, you can access the user's identity details inside a controller action. For instance, by setting a breakpoint inside the `HomeController.Index` action method, you can view the `User.claims` details. By having the user signed-in, you can make authorization decisions. For more information, see *Authorization*.

As a registered user, you can log in to the web app by clicking the **Log in** link. When a registered user logs in, the `Login` action of the `AccountController` is called. Then, the **Login** action signs in the user using the `PasswordSignInAsync` method contained in the `Login` action.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(model.Email, model.Password, model
        if (result.Succeeded)
        {
            _logger.LogInformation(1, "User logged in.");
            return RedirectToLocal(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToAction(nameof(SendCode), new { ReturnUrl = returnUrl, RememberMe =
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning(2, "User account locked out.");
            return View("Lockout");
        }
```

```
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return View(model);
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

4. Log off.

   Clicking the **Log off** link calls the `LogOff` action in the account controller.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> LogOff()
{
    await _signInManager.SignOutAsync();
    _logger.LogInformation(4, "User logged out.");
    return RedirectToAction(nameof(HomeController.Index), "Home");
}
```

   The code above shows the `SignInManager.SignOutAsync` method. The `SignOutAsync` method clears the users claims stored in a cookie.

5. View the database.

   After stopping the application, view the user database from Visual Studio by selecting **View** -> **SQL Server Object Explorer**. Then, expand the following within the **SQL Server Object Explorer**:

   - (localdb)MSSQLLocalDB

   - Databases

   - aspnet5-*<the name of your application>*

   - Tables

   Next, right-click the **dbo.AspNetUsers** table and select **View Data** to see the properties of the user you created.
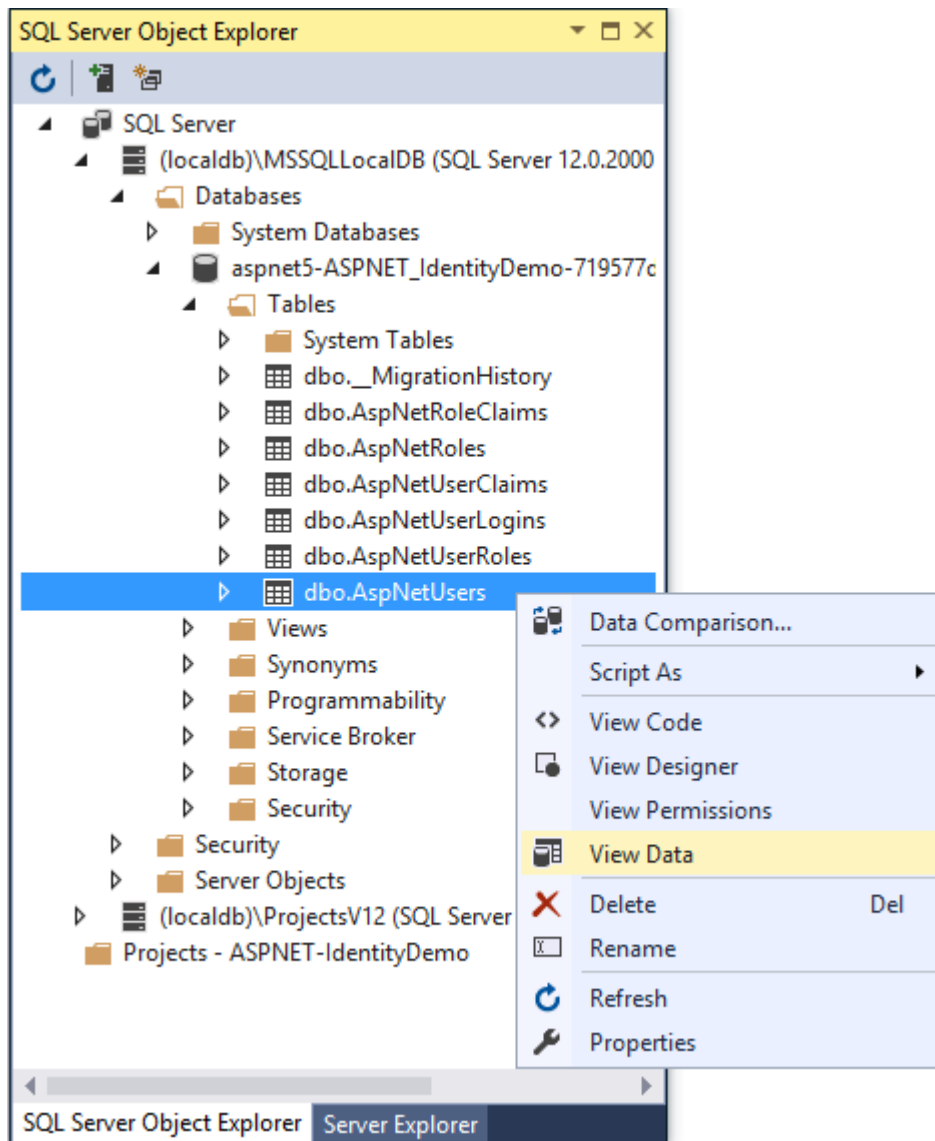
Identity Components

The primary reference assembly for the identity system is `Microsoft.AspNetCore.Identity`. This package contains the core set of interfaces for ASP.NET Core Identity.
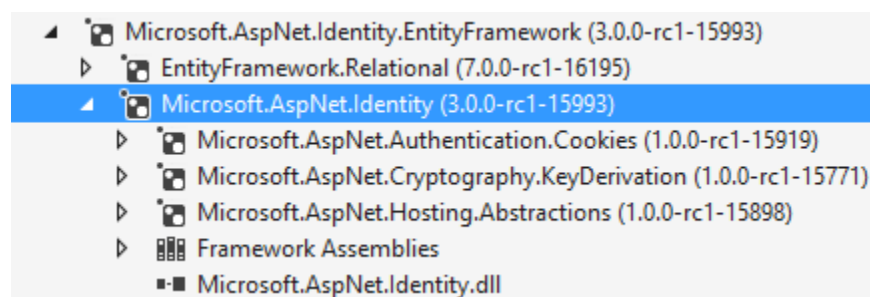
These dependencies are needed to use the identity system in ASP.NET Core applications:

- `EntityFramework.SqlServer` - Entity Framework is Microsoft's recommended data access technology for relational databases.

- `Microsoft.AspNetCore.Authentication.Cookies` - Middleware that enables an application to use cookie based authentication, similar to ASP.NET's Forms Authentication.

- `Microsoft.AspNetCore.Cryptography.KeyDerivation` - Utilities for key derivation.

- `Microsoft.AspNetCore.Hosting.Abstractions` - Hosting abstractions.

Migrating to ASP.NET Core Identity

For additional information and guidance on migrating your existing identity store see *Migrating Authentication and Identity*

Next Steps

- *Migrating Authentication and Identity*

- *Account Confirmation and Password Recovery*

- *Two-factor authentication with SMS*

- *Enabling authentication using Facebook, Google and other external providers*

Enabling authentication using Facebook, Google and other external providers

By Rick Anderson and Pranav Rastogi

This tutorial shows you how to build an ASP.NET Core app that enables users to log in using OAuth 2.0 with credentials from an external authentication provider, such as Facebook, Twitter, LinkedIn, Microsoft, and Google. For simplicity, this tutorial focuses on working with credentials from Facebook and Google.

Enabling these credentials in your web sites provides a significant advantage because millions of users already have accounts with these external providers. These users may be more inclined to sign up for your site if they do not have to create and remember new credentials.

Sections:

- *Create a New ASP.NET Core Project*
- *Creating the app in Facebook*
- *Use SecretManager to store Facebook AppId and AppSecret*
- *Enable Facebook middleware*
- *Login with Facebook*
- *Optionally set password*
- *Next steps*

Create a New ASP.NET Core Project

: The tutorial requires the latest version of Visual Studio 2015 and ASP.NET Core.

- In Visual Studio, create a New Project (from the Start Page, or via **File > New > Project**)
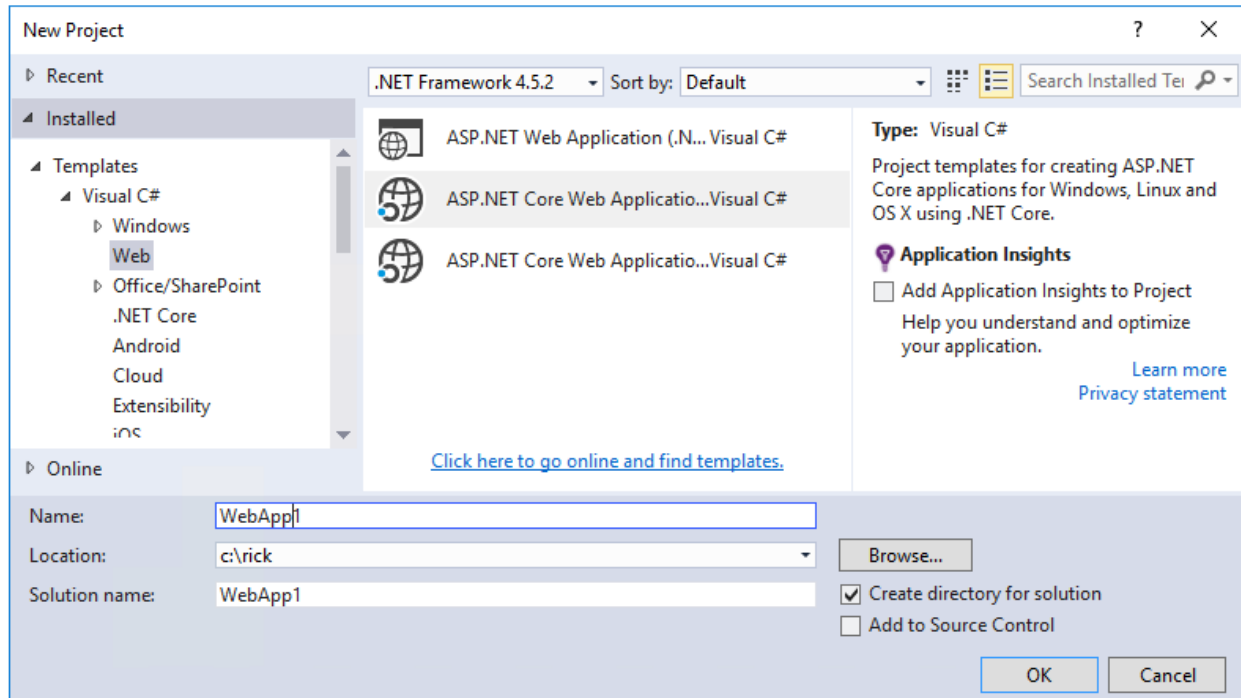
- Tap **Web Application** and verify **Authentication** is set to **Individual User Accounts**
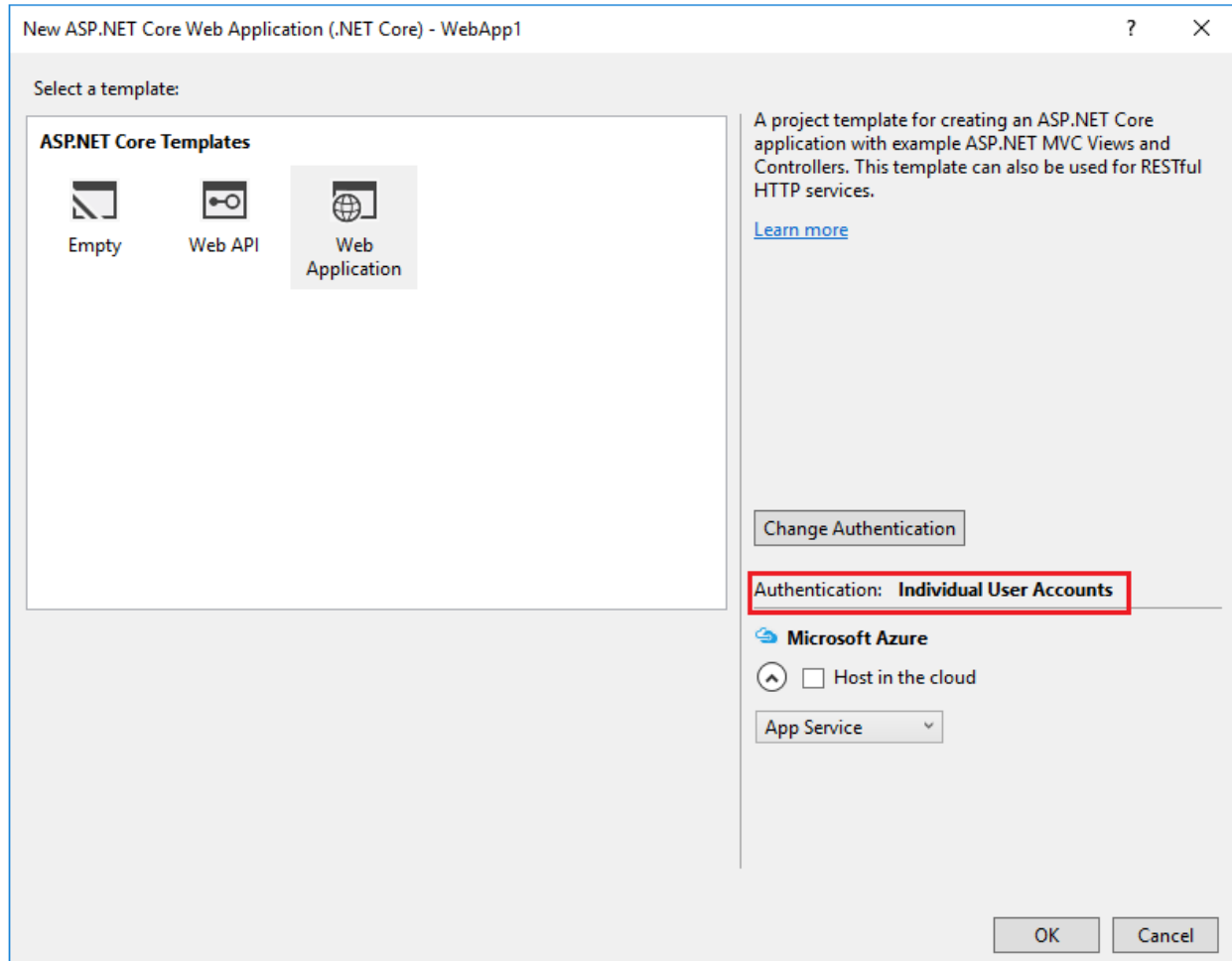
- Enable SSL

    - In solution explorer, right click the project and select **Properties**

    - On the left pane, tap **Debug**

    - Check **Enable SSL**

    - Copy the SSL URL and paste it into the **App URL**
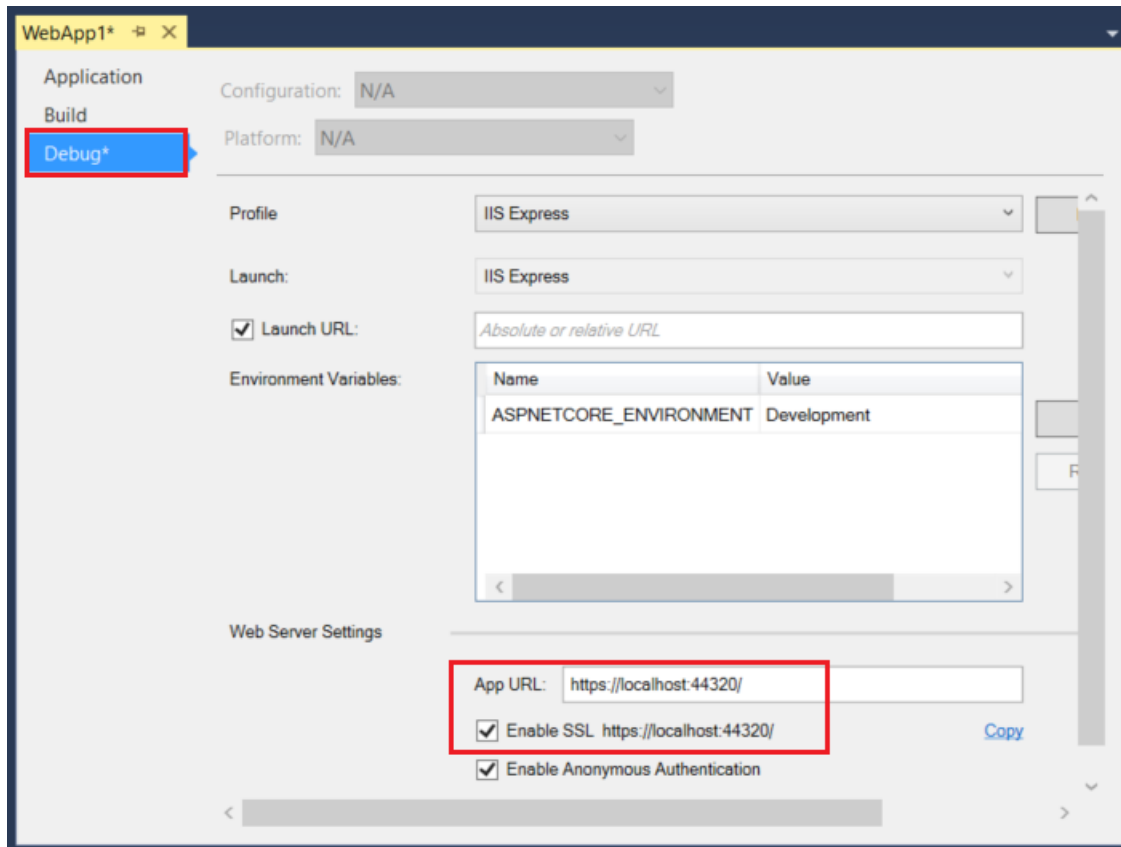
- Require SSL. Add the following code to `ConfigureServices` in `Startup`:

```
services.Configure<MvcOptions>(options =>
{
    options.Filters.Add(new RequireHttpsAttribute ());
});
```

- Test the app

Creating the app in Facebook

Each of the OAuth2 providers require provider specific keys to enable OAuth2.

- Navigate to https://developers.facebook.com/apps and log in.

- If you aren't already registered as a Facebook developer, click *Register as a Developer* and follow the directions to register.

- Tap **Add a New App**

- Select **Website** from the platform choices.

- Tap **Skip and Create App ID**
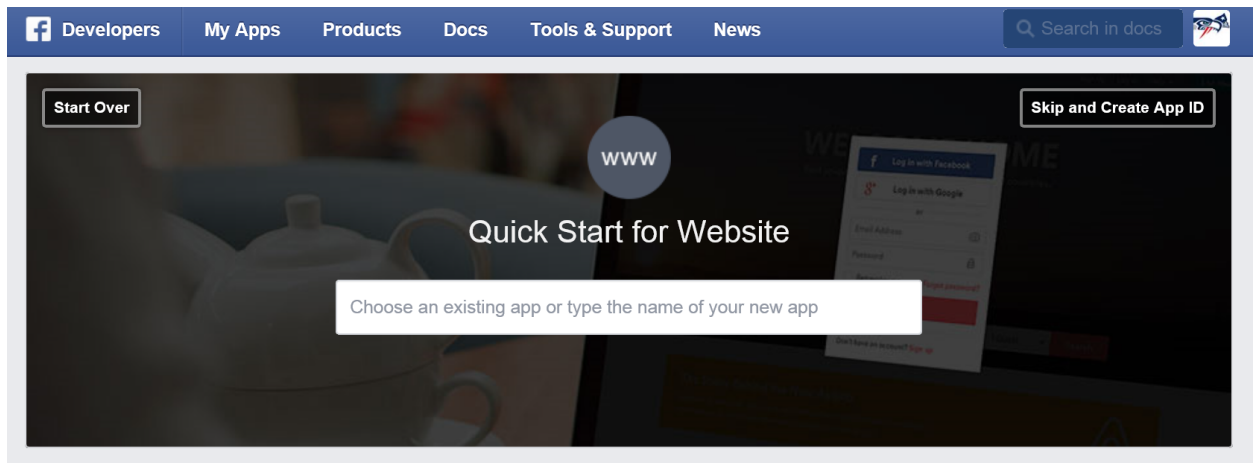
- Enter a display name, category, contact email and tap **Create App ID**.
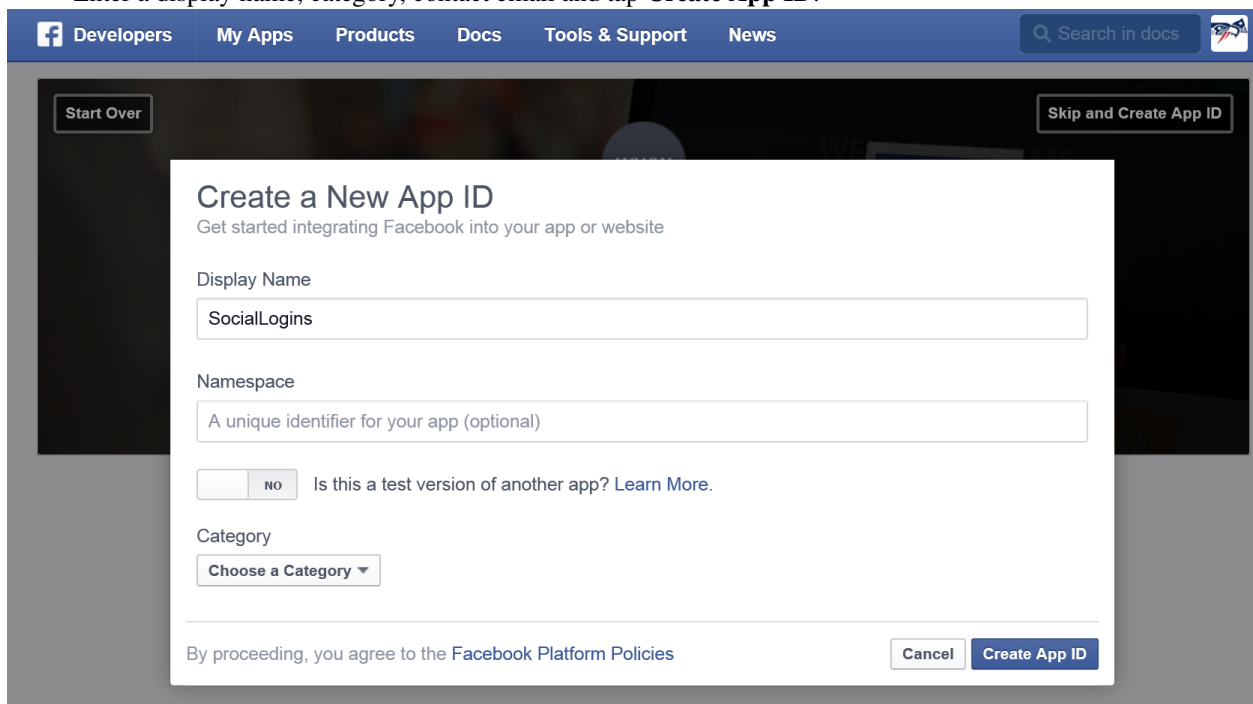
- Tap **Settings** from the left menu bar.
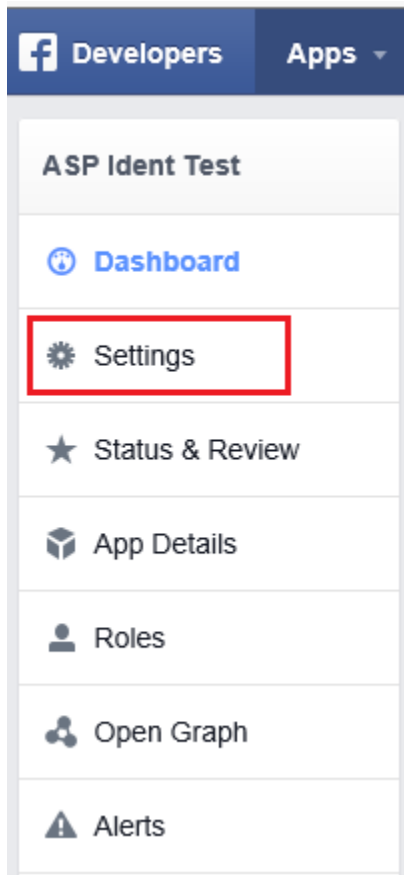
- On the **Basic** settings section of the page select Add Platform to specify that you are adding a website app.
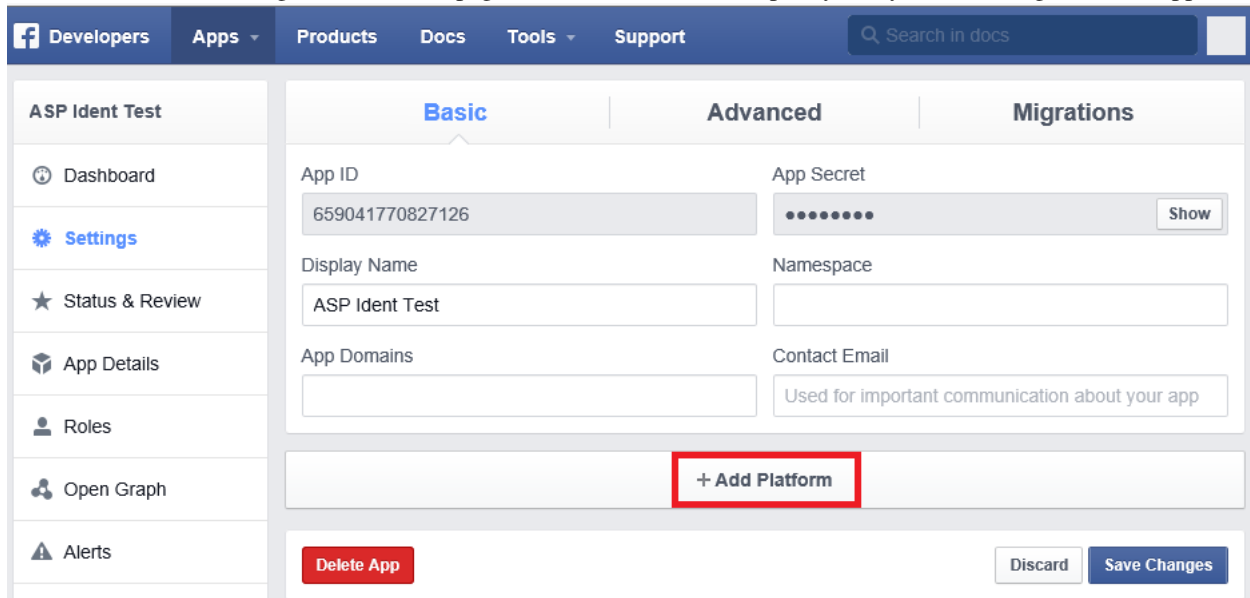
- Select Website from the platform choices.

- Add your Site URL (https://localhost:44320/)

- Make a note of your App ID and your App Secret so that you can add both into your ASP.NET Core app later in this tutorial. Also, Add your Site URL (https://localhost:44300/) to test your application.

Use SecretManager to store Facebook AppId and AppSecret

The project created has code in Startup which reads the configuration values from a secret store. As a best practice, it is not recommended to store the secrets in a configuration file in the application since they can be checked into source control which may be publicly accessible.

Follow these steps to add the Facebook AppId and AppSecret to the Secret Manager:

- Install the *Secret Manager tool*.
- Set the Facebook AppId:

```
dotnet user-secrets set Authentication:Facebook:AppId <app-Id>
```

- Set the Facebook AppSecret:

```
dotnet user-secrets set Authentication:Facebook:AppSecret <app-secret>
```

The following code reads the configuration values stored by the *Secret Manager*.

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);
```

```
    if (env.IsDevelopment())
    {
        // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID
        builder.AddUserSecrets();
    }

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

Enable Facebook middleware

Note: You will need to use NuGet to install the Microsoft.AspNetCore.Authentication.Facebook package if it hasn't already been installed.

Add the Facebook middleware in the `Configure` method in `Startup`:

```csharp
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseIdentity();

    app.UseFacebookAuthentication(new FacebookOptions()
    {
        AppId = Configuration["Authentication:Facebook:AppId"],
        AppSecret = Configuration["Authentication:Facebook:AppSecret"]
    });

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

Login with Facebook

Run your application and click Login. You will see an option for Facebook.

When you click on Facebook, you will be redirected to Facebook for authentication.

Once you enter your Facebook credentials, then you will be redirected back to the Web site where you can set your email.

You are now logged in using your Facebook credentials.
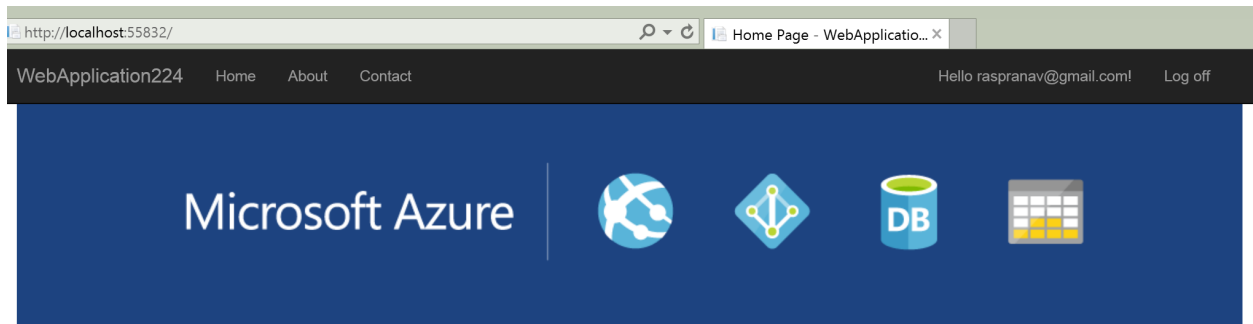
Optionally set password

When you register with an external login provider, you do not have a password registered with the app. This alleviates you from creating and remembering a password for the site, but it also makes you dependent on the external login provider. If the external login provider is unavailable, you won't be able to log in to the web site.

To create a password and login using your email that you set during the login process with external providers:

- Tap the **Hello <email alias>** link at the top right corner to navigate to the **Manage** view.
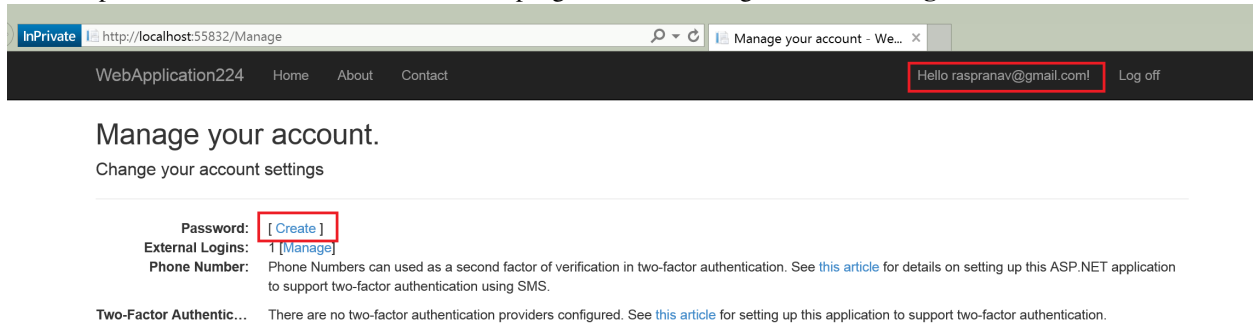
- Tap **Create**
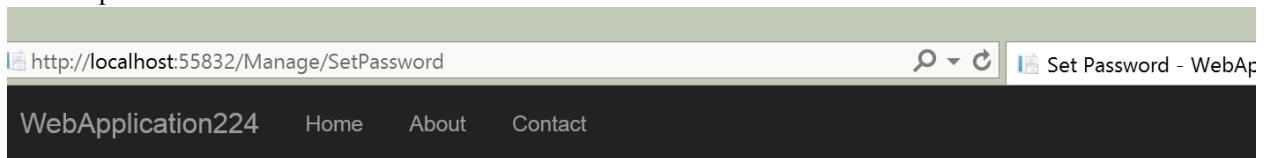
- Set a valid password and you can use this to login with your email

---

Next steps

- This article showed how you can authenticate with Facebook. You can follow a similar approach to authenticate with Microsoft Account, Twitter, Google and other providers.

- Once you publish your Web site to Azure Web App, you should reset the AppSecret in the Facebook developer portal.

- Set the Facebook AppId and AppSecret as application setting in the Azure Web App portal. The configuration system is setup to read keys from environment variables.

Account Confirmation and Password Recovery

By Rick Anderson

This tutorial shows you how to build an ASP.NET Core app with email confirmation and password reset support.

> **Sections:**
>
> - *Create a New ASP.NET Core Project*
> - *Require SSL*
> - *Require email confirmation*
> - *Enable account confirmation and password recovery*
> - *Register, confirm email, and reset password*
> - *Require email confirmation before login*
> - *Combine social and local login accounts*

Create a New ASP.NET Core Project

: The tutorial requires Visual Studio 2015 updated 2 and ASP.NET Core RC2 or higher.

- In Visual Studio, create a New Project (from the Start Page, or via **File > New > Project**)
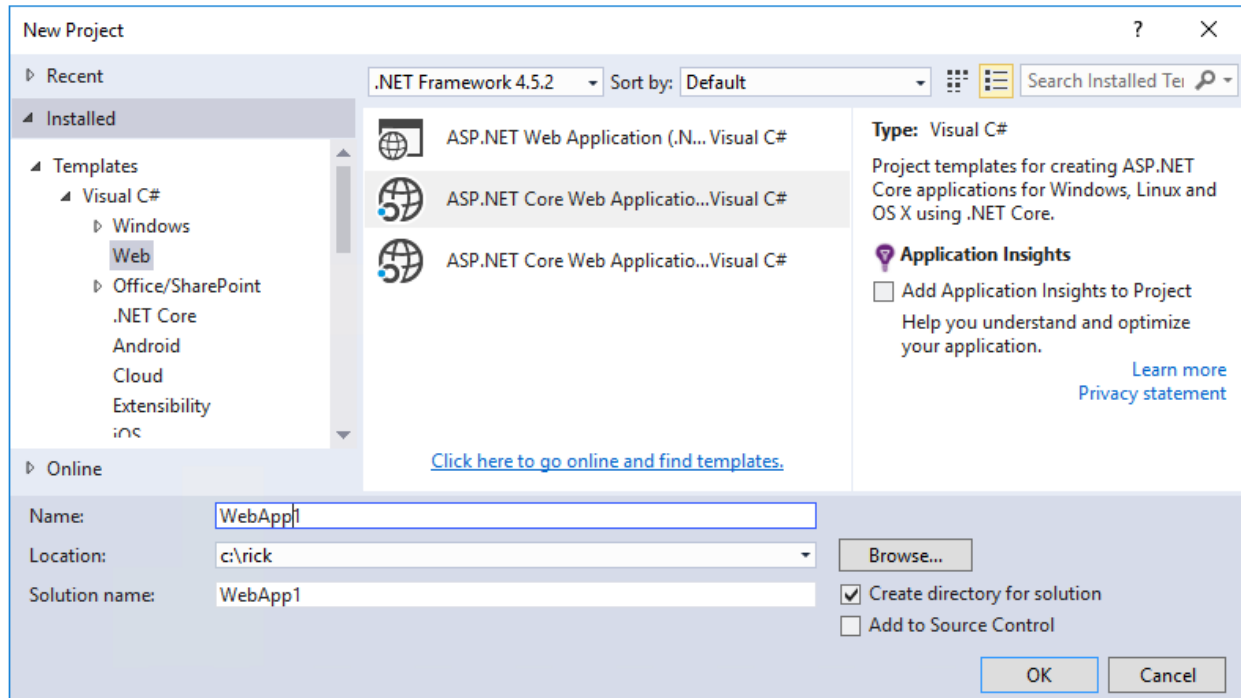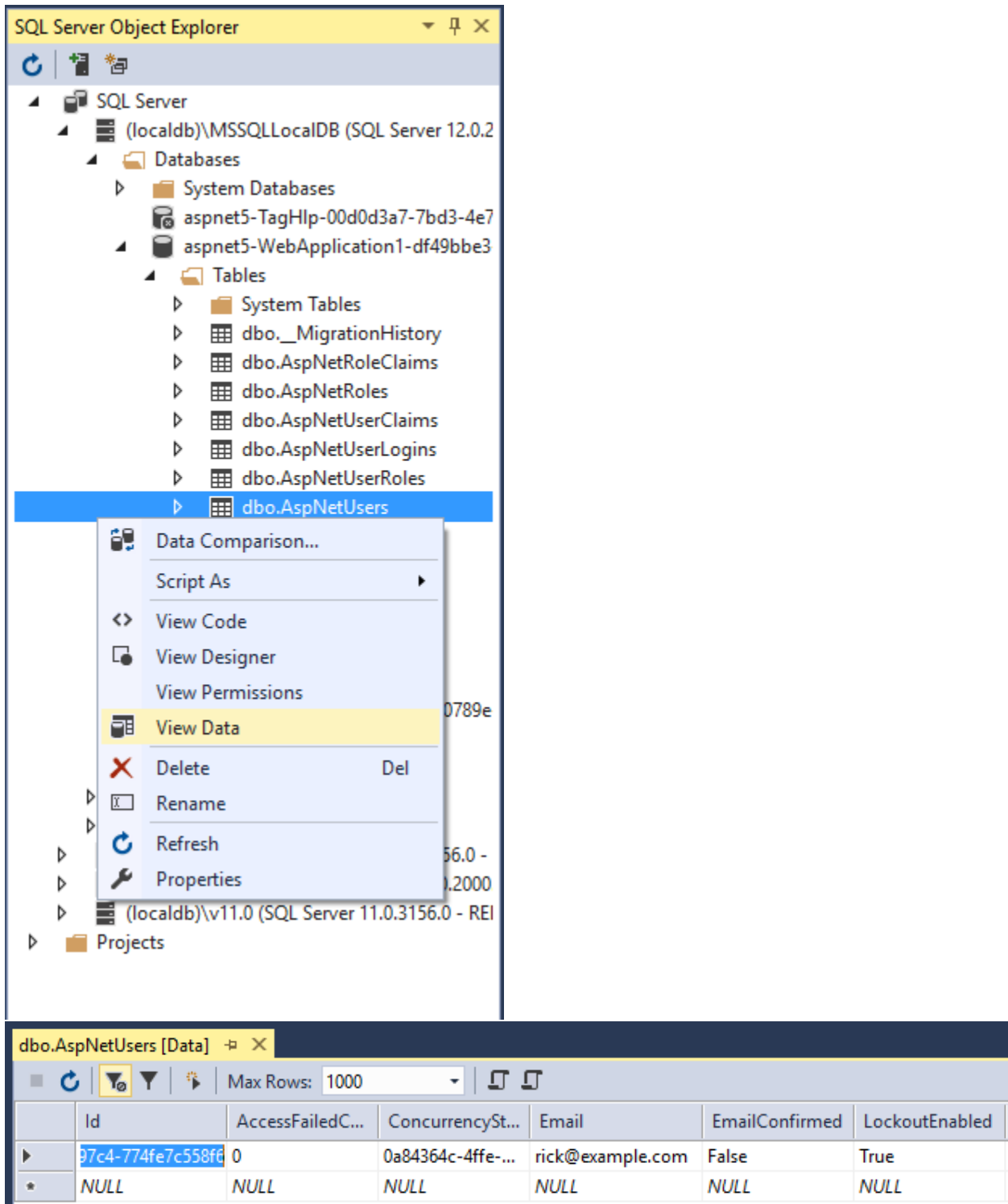
- Tap **Web Application** and verify **Authentication** is set to **Individual User Accounts**
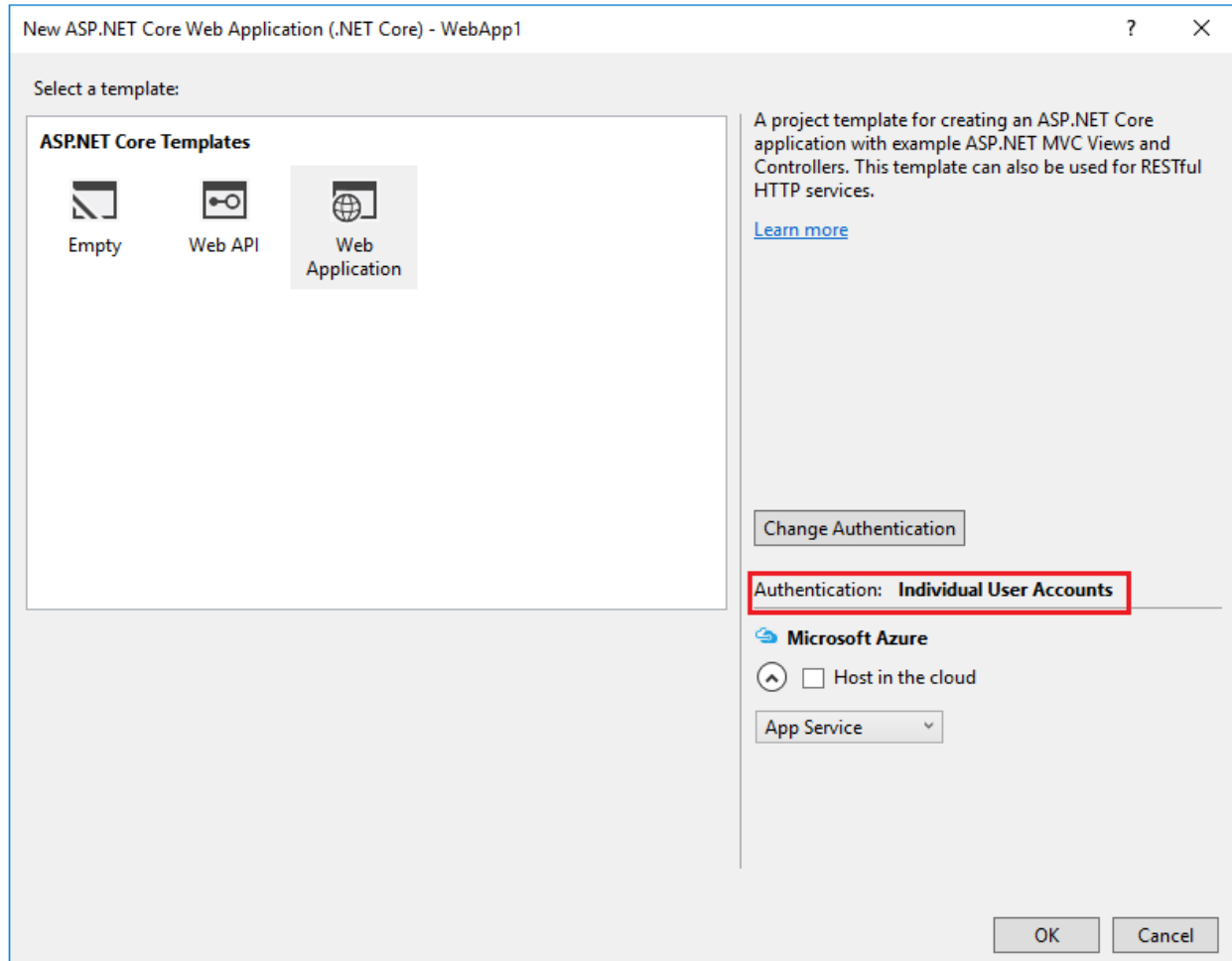
Run the app and then click on the **Register** link and register a user. At this point, the only validation on the email is with the [EmailAddress] attribute. After you submit the registration, you are logged into the app. Later in the tutorial we'll change this so new users cannot log in until their email has been validated.

In **SQL Server Object Explorer** (SSOX), navigate to **(localdb)MSSQLLocalDB(SQL Server 12)**. Right click on **dbo.AspNetUsers > View Data**:

Note the `EmailConfirmed` field is `False`.

Right-click on the row and from the context menu, select **Delete**. You might want to use this email again in the next step, when the app sends a confirmation email. Deleting the email alias now will make it easier in the following steps.

Require SSL

In this section we'll set up our Visual Studio project to use SSL and our project to require SSL.

Enable SSL in Visual Studio

- In solution explorer, right click the project and select **Properties**
- On the left pane, tap **Debug**
- Check **Enable SSL**
- Copy the SSL URL and paste it into the **App URL**
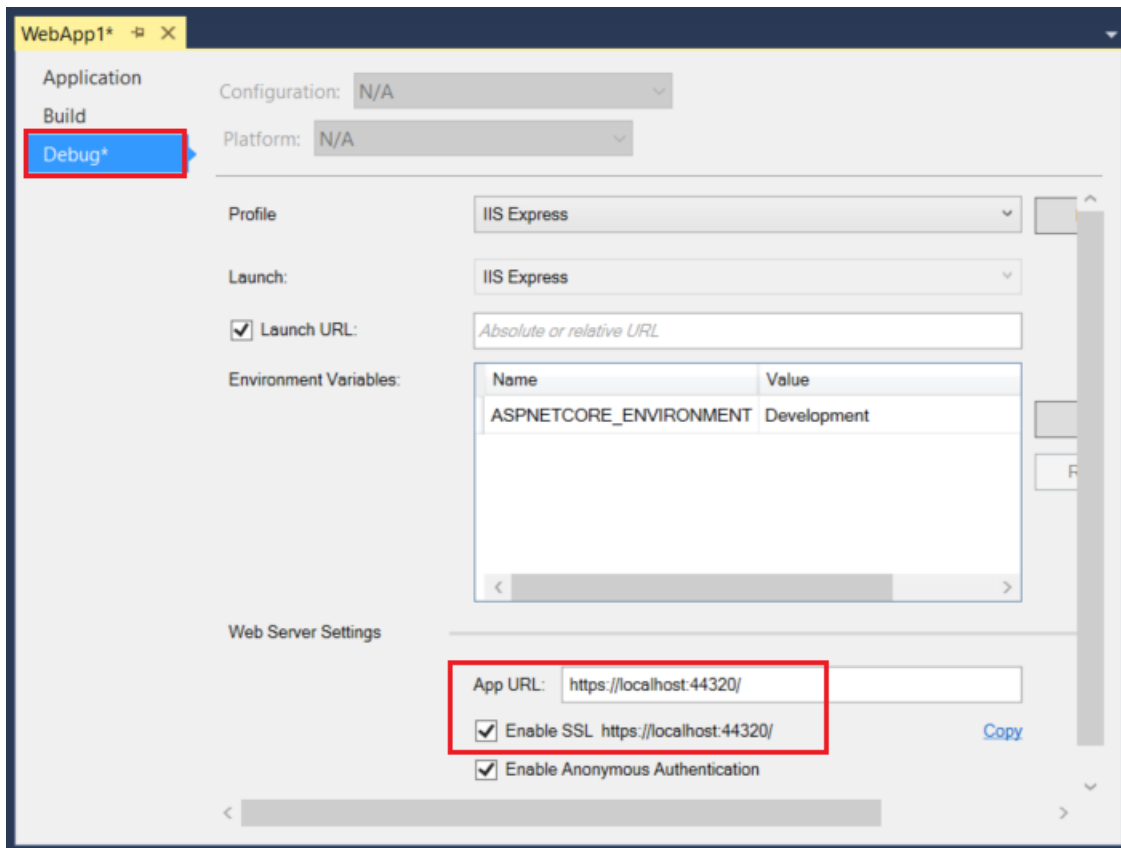
- Add the following code to `ConfigureServices` in `Startup`:

```
services.Configure<MvcOptions>(options =>
{
    options.Filters.Add(new RequireHttpsAttribute ());
});
```

Add the `[RequireHttps]` attribute to each controller. The `[RequireHttps]` attribute will redirect all HTTP GET requests to HTTPS GET and will reject all HTTP POSTs. A security best practice is to use HTTPS for all requests.

```
[RequireHttps]
public class HomeController : Controller
```

Require email confirmation

It's a best practice to confirm the email of a new user registration to verify they are not impersonating someone else (that is, they haven't registered with someone else's email). Suppose you had a discussion forum, you would want to prevent "bob@example.com" from registering as "joe@contoso.com". Without email confirmation, "joe@contoso.com" could get unwanted email from your app. Suppose Bob accidentally registered as "bib@example.com" and hadn't noticed it, he wouldn't be able to use password recovery because the app doesn't have his correct email. Email confirmation provides only limited protection from bots and doesn't provide protection from determined spammers who have many working email aliases they can use to register.

You generally want to prevent new users from posting any data to your web site before they have been confirmed by email, an SMS text message, or another mechanism. In the sections below, we will enable email confirmation and modify the code to prevent newly registered users from logging in until their email has been confirmed.

Configure email provider   We'll use the *Options pattern* to access the user account and key settings. For more information, see *configuration*.

- Create a class to fetch the secure email key. For this sample, the AuthMessageSenderOptions class is created in the *Services/AuthMessageSenderOptions.cs* file.

```
public class AuthMessageSenderOptions
{
    public string SendGridUser { get; set; }
    public string SendGridKey { get; set; }
}
```

Set the SendGridUser and SendGridKey with the *secret-manager tool*. For example:

```
C:\WebApplication3\src\WebApplication3>dotnet user-secrets set SendGridUser RickAndMSFT
info: Successfully saved SendGridUser = RickAndMSFT to the secret store.
```

On Windows, Secret Manager stores your keys/value pairs in a *secrets.json* file in the %APP-DATA%/Microsoft/UserSecrets/**<userSecretsId>** directory. The **userSecretsId** directory can be found in your *project.json* file. For this example, the first few lines of the *project.json* file are shown below:

```
{
    "webroot": "wwwroot",
    "userSecretsId": "aspnet-WebApplication3-f1645c1b-3962-4e7f-99b2-4fb292b6dade",
    "version": "1.0.0-*",

    "dependencies": {
```

At this time, the contents of the *secrets.json* file are not encrypted. The *secrets.json* file is shown below (the sensitive keys have been removed.)

```
{
  "SendGridUser": "RickAndMSFT",
  "SendGridKey": "",
  "Authentication:Facebook:AppId": "",
  "Authentication:Facebook:AppSecret": ""
}
```

Configure startup to use `AuthMessageSenderOptions` Add the dependecy `Microsoft.Extensions.Options.ConfigurationExtensions` in the project.json file.

Add `AuthMessageSenderOptions` to the service container at the end of the `ConfigureServices` method in the *Startup.cs* file:

```
// Add application services.
services.AddTransient<IEmailSender, AuthMessageSender>();
services.AddTransient<ISmsSender, AuthMessageSender>();
services.Configure<AuthMessageSenderOptions>(Configuration);
```

Configure the `AuthMessageSender` class This tutorial shows how to add email notification through SendGrid, but you can send email using SMTP and other mechanisms.

- Install the SendGrid.NetCore NuGet package. From the Package Manager Console, enter the following the following command:

```
Install-Package SendGrid.NetCore –Pre
```

: SendGrid.NetCore package is a prerelease version , to install it is necessary to use -Pre option on Install-Package.

- Follow the instructions Create a SendGrid account to register for a free SendGrid account.

- Add code in *Services/MessageServices.cs* similar to the following to configure SendGrid

```csharp
public class AuthMessageSender : IEmailSender, ISmsSender
{
    public AuthMessageSender(IOptions<AuthMessageSenderOptions> optionsAccessor)
    {
        Options = optionsAccessor.Value;
    }

    public AuthMessageSenderOptions Options { get; } //set only via Secret Manager

    public Task SendEmailAsync(string email, string subject, string message)
    {
        // Plug in your email service here to send an email.
        var myMessage = new SendGrid.SendGridMessage();
        myMessage.AddTo(email);
        myMessage.From = new System.Net.Mail.MailAddress("Joe@contoso.com", "Joe Smith");
        myMessage.Subject = subject;
        myMessage.Text = message;
        myMessage.Html = message;
        var credentials = new System.Net.NetworkCredential(
            Options.SendGridUser,
            Options.SendGridKey);
        // Create a Web transport for sending email.
        var transportWeb = new SendGrid.Web(credentials);
        // Send the email.
        if (transportWeb != null)
        {
            return transportWeb.DeliverAsync(myMessage);
        }
        else
        {
```

```
            return Task.FromResult(0);
        }
    }

    public Task SendSmsAsync(string number, string message)
    {
        // Plug in your SMS service here to send a text message.
        return Task.FromResult(0);
    }
}
```

Enable account confirmation and password recovery

The template already has the code for account confirmation and password recovery. Follow these steps to enable it:

- Find the [HttpPost] Register method in the *AccountController.cs* file.

- Uncomment the code to enable account confirmation.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // For more information on how to enable account confirmation and password reset please v
            // Send an email with this link
            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code = co
            await _emailSender.SendEmailAsync(model.Email, "Confirm your account",
                $"Please confirm your account by clicking this link: <a href='{callbackUrl}'>link</a>
            //await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation(3, "User created a new account with password.");
            return RedirectToLocal(returnUrl);

        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Note: We're also preventing a newly registered user from being automatically logged on by commenting out the following line:

```
//await _signInManager.SignInAsync(user, isPersistent: false);
```

- Enable password recovery by uncommenting the code in the ForgotPassword action in the *Controllers/AccountController.cs* file.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = await _userManager.FindByNameAsync(model.Email);
        if (user == null || !(await _userManager.IsEmailConfirmedAsync(user)))
        {
            // Don't reveal that the user does not exist or is not confirmed
            return View("ForgotPasswordConfirmation");
        }

        // For more information on how to enable account confirmation and password reset please visit
        // Send an email with this link
        var code = await _userManager.GeneratePasswordResetTokenAsync(user);
        var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = code
        await _emailSender.SendEmailAsync(model.Email, "Reset Password",
            $"Please reset your password by clicking here: <a href='{callbackUrl}'>link</a>");
        return View("ForgotPasswordConfirmation");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Uncomment the highlighted `ForgotPassword` from in the *Views/Account/ForgotPassword.cshtml* view file.

```
@model ForgotPasswordViewModel
@{
    ViewData["Title"] = "Forgot your password?";
}

<h2>@ViewData["Title"]</h2>
<p>
    For more information on how to enable reset password please see this <a href="http://go.microsoft
</p>

<form asp-controller="Account" asp-action="ForgotPassword" method="post" class="form-horizontal">
    <h4>Enter your email.</h4>
    <hr />
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Submit</button>
        </div>
    </div>
</form>
```

```
@section Scripts {
    @{ await Html.RenderPartialAsync("_ValidationScriptsPartial"); }
}
```

Register, confirm email, and reset password

In this section, run the web app and show the account confirmation and password recovery flow.

- Run the application and register a new user

- Check your email for the account confirmation link. If you don't get the email notification:
    - Check the SendGrid web site to verify your sent mail messages.
    - Check your spam folder.
    - Try another email alias on a different email provider (Microsoft, Yahoo, Gmail, etc.)
    - In SSOX, navigate to **dbo.AspNetUsers** and delete the email entry and try again.
- Click the link to confirm your email.
- Log in with your email and password.
- Log off.

Test password reset

- Login and select **Forgot your password?**

- Enter the email you used to register the account.

- An email with a link to reset your password will be sent. Check your email and click the link to reset your password. After your password has been successfully reset, you can login with your email and new password.

Require email confirmation before login

With the current templates, once a user completes the registration form, they are logged in (authenticated). You generally want to confirm their email before logging them in. In the section below, we will modify the code to require new users have a confirmed email before they are logged in. Update the `[HttpPost]` `Login` action in the *AccountController.cs* file with the following highlighted changes.

```csharp
//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        // Require the user to have a confirmed email before they can log on.
        var user = await _userManager.FindByNameAsync(model.Email);
        if (user != null)
        {
            if (!await _userManager.IsEmailConfirmedAsync(user))
            {
                ModelState.AddModelError(string.Empty, "You must have a confirmed email to log in.");
                return View(model);
            }
        }

        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(model.Email, model.Password, model.Reme
        if (result.Succeeded)
        {
            _logger.LogInformation(1, "User logged in.");
            return RedirectToLocal(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToAction(nameof(SendCode), new { ReturnUrl = returnUrl, RememberMe = model
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning(2, "User account locked out.");
            return View("Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return View(model);
```

```
        }
    }
```

: A security best practice is to not use production secrets in test and development. If you publish the app to Azure, you can set the SendGrid secrets as application settings in the Azure Web App portal. The configuration system is setup to read keys from environment variables.

Combine social and local login accounts

To complete this section, you must first enable an external authentication provider. See *Enabling authentication using Facebook, Google and other external providers*.

You can combine local and social accounts by clicking on your email link. In the following sequence "RickAndMSFT@gmail.com" is first created as a local login, but you can create the account as a social login first, then add a local login.
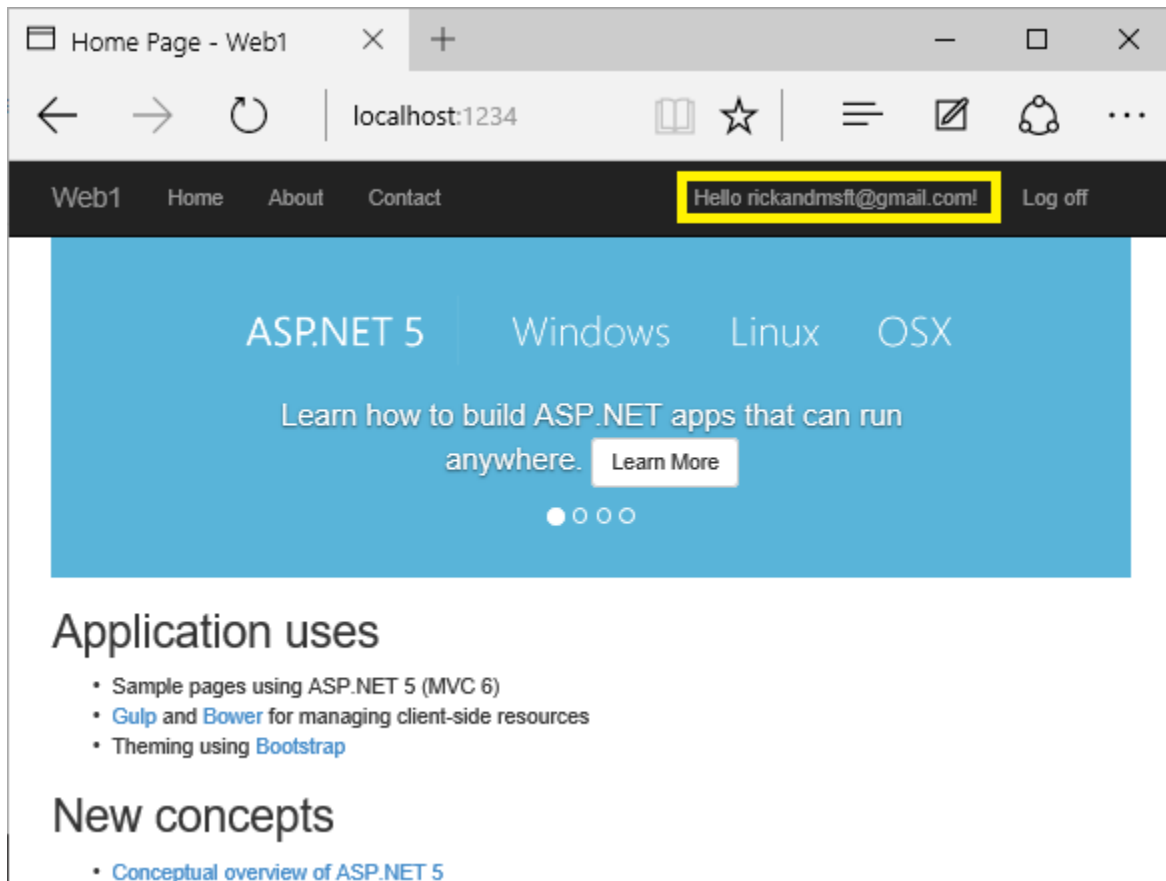
Click on the **Manage** link. Note the 0 external (social logins) associated with this account.
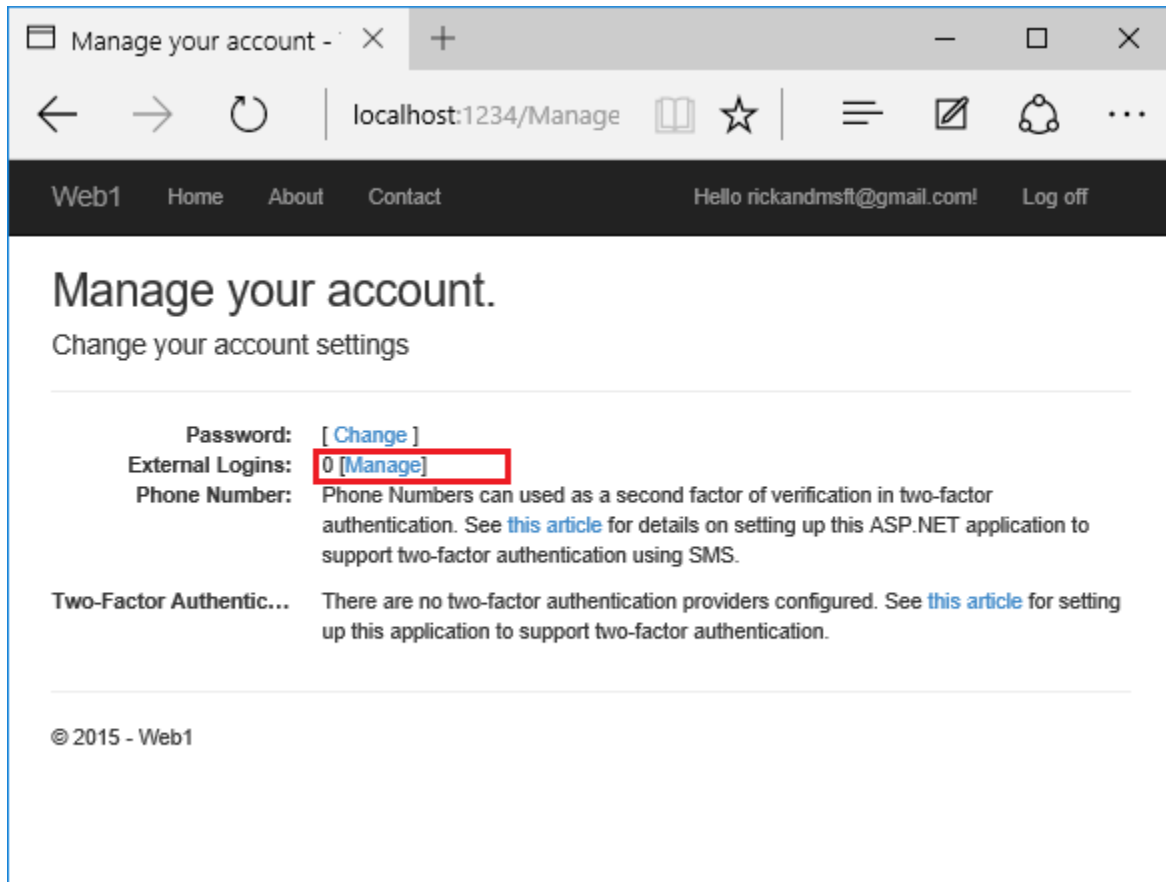
Click the link to another login service and accept the app requests. In the image below, Facebook is the external authentication provider:
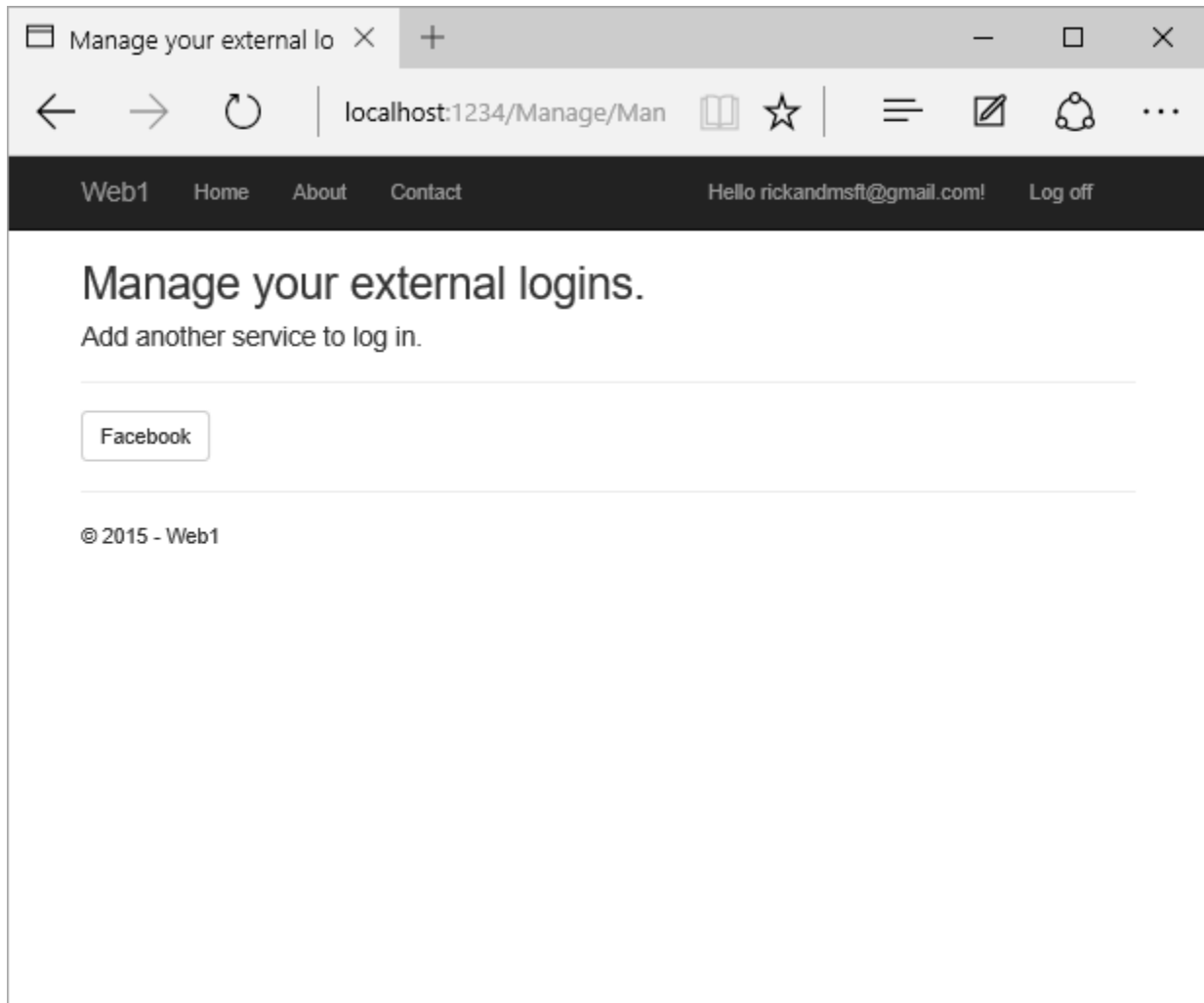
The two accounts have been combined. You will be able to log on with either account. You might want your users to add local accounts in case their social log in authentication service is down, or more likely they have lost access to their social account.

Two-factor authentication with SMS

By Rick Anderson

This tutorial will show you how to set up two-factor authentication (2FA) using SMS. Twilio is used, but you can use any other SMS provider. We recommend you complete *Account Confirmation and Password Recovery* before starting this tutorial.

Sections:

- *Create a new ASP.NET Core project*
- *Setup up SMS for two-factor authentication with Twilio*
- *Enable two-factor authentication*
- *Log in with two-factor authentication*
- *Account lockout for protecting against brute force attacks*
- *Debugging Twilio*

Create a new ASP.NET Core project

Create a new ASP.NET Core web app with individual user accounts.
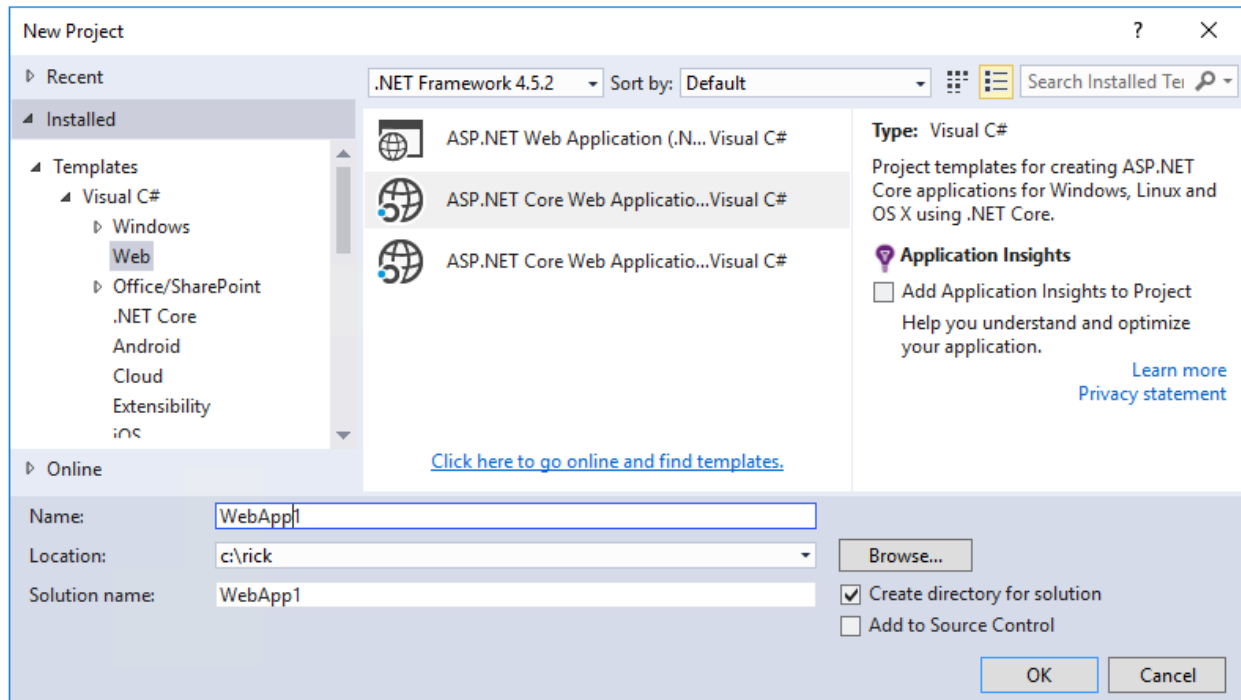
After you create the project, follow the instructions in *Account Confirmation and Password Recovery* to set up and require SSL.

Setup up SMS for two-factor authentication with Twilio

- Create a Twilio account.

- On the **Dashboard** tab of your Twilio account, note the **Account SID** and **Authentication token**. Note: Tap **Show API Credentials** to see the Authentication token.

- On the **Numbers** tab, note the Twilio phone number.

- Install the Twilio NuGet package. From the Package Manager Console (PMC), enter the following the following command:

```
Install-Package Twilio
```

- Add code in the *Services/MessageServices.cs* file to enable SMS.

```csharp
public class AuthMessageSender : IEmailSender, ISmsSender
{
    public AuthMessageSender(IOptions<AuthMessageSMSSenderOptions> optionsAccessor)
    {
        Options = optionsAccessor.Value;
    }

    public AuthMessageSMSSenderOptions Options { get; }  // set only via Secret Manager

    public Task SendEmailAsync(string email, string subject, string message)
```

```
    {
        // Plug in your email service here to send an email.
        return Task.FromResult(0);
    }

    public Task SendSmsAsync(string number, string message)
    {
        var twilio = new Twilio.TwilioRestClient(
            Options.SID,            // Account Sid from dashboard
            Options.AuthToken);     // Auth Token

        var result = twilio.SendMessage(Options.SendNumber, number, message);
        // Use the debug output for testing without receiving a SMS message.
        // Remove the Debug.WriteLine(message) line after debugging.
        // System.Diagnostics.Debug.WriteLine(message);
        return Task.FromResult(0);
    }
}
```

: Twilio does not yet support .NET Core. To use Twilio from your application you need to either target the full .NET Framework or you can call the Twilio REST API to send SMS messages.

: You can remove // line comment characters from the `System.Diagnostics.Debug.WriteLine(message);` line to test the application when you can't get SMS messages. A better approach to logging is to use the built in *logging*.

Configure the SMS provider key/value   We'll use the *Options pattern* to access the user account and key settings. For more information, see *configuration*.

- Create a class to fetch the secure SMS key. For this sample, the `AuthMessageSMSSenderOptions` class is created in the *Services/AuthMessageSMSSenderOptions.cs* file.

```
public class AuthMessageSMSSenderOptions
{
    public string SID { get; set; }
    public string AuthToken { get; set; }
    public string SendNumber { get; set; }
}
```

Set `SID`, `AuthToken`, and `SendNumber` with the *secret-manager tool*. For example:

```
C:/WebSMS/src/WebApplication1>dotnet user-secrets set SID abcdefghi
info: Successfully saved SID = abcdefghi to the secret store.
```

Configure startup to use `AuthMessageSMSSenderOptions`   Add `AuthMessageSMSSenderOptions` to the service container at the end of the `ConfigureServices` method in the *Startup.cs* file:

```
    // Register application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
```

```
    services.Configure<AuthMessageSMSSenderOptions>(Configuration);
}
```

Enable two-factor authentication

- Open the *Views/Manage/Index.cshtml* Razor view file.

- Uncomment the phone number markup which starts at

  ```
  @*@(Model.PhoneNumber ??  "None")
  ```

- Uncomment the `Model.TwoFactor` markup which starts at

  ```
  @*@if (Model.TwoFactor)
  ```

- Comment out or remove the `<p>There are no two-factor authentication providers configured.` markup.

  The completed code is shown below:

```
<dt>Phone Number:</dt>
<dd>
    <p>
        Phone Numbers can used as a second factor of verification in two-factor authentication.
        See <a href="http://go.microsoft.com/fwlink/?LinkID=532713">this article</a>
        for details on setting up this ASP.NET application to support two-factor authentication
    </p>
    @(Model.PhoneNumber ?? "None") [
        @if (Model.PhoneNumber != null)
        {
            <a asp-controller="Manage" asp-action="AddPhoneNumber">Change</a>
              @:  | 
              <a asp-controller="Manage" asp-action="RemovePhoneNumber">Remove</a>
        }
        else
        {
            <a asp-controller="Manage" asp-action="AddPhoneNumber">Add</a>
        }
        ]
</dd>

<dt>Two-Factor Authentication:</dt>
<dd>
    @*<p>
        There are no two-factor authentication providers configured. See <a href="http://go.micr
        for setting up this application to support two-factor authentication.
    </p>*@
    @if (Model.TwoFactor)
        {
            <form asp-controller="Manage" asp-action="DisableTwoFactorAuthentication" method="po
                <text>
                    Enabled
                    <button type="submit" class="btn btn-link">Disable</button>
                </text>
            </form>
        }
        else
        {
```

```
                <form asp-controller="Manage" asp-action="EnableTwoFactorAuthentication" method="pos
                    <text>
                        Disabled
                        <button type="submit" class="btn btn-link">Enable</button>
                    </text>
                </form>
            }
        </dd>
```

Log in with two-factor authentication

- Run the app and register a new user

- Tap on your user name, which activates the `Index` action method in Manage controller. Then tap the phone number **Add** link.
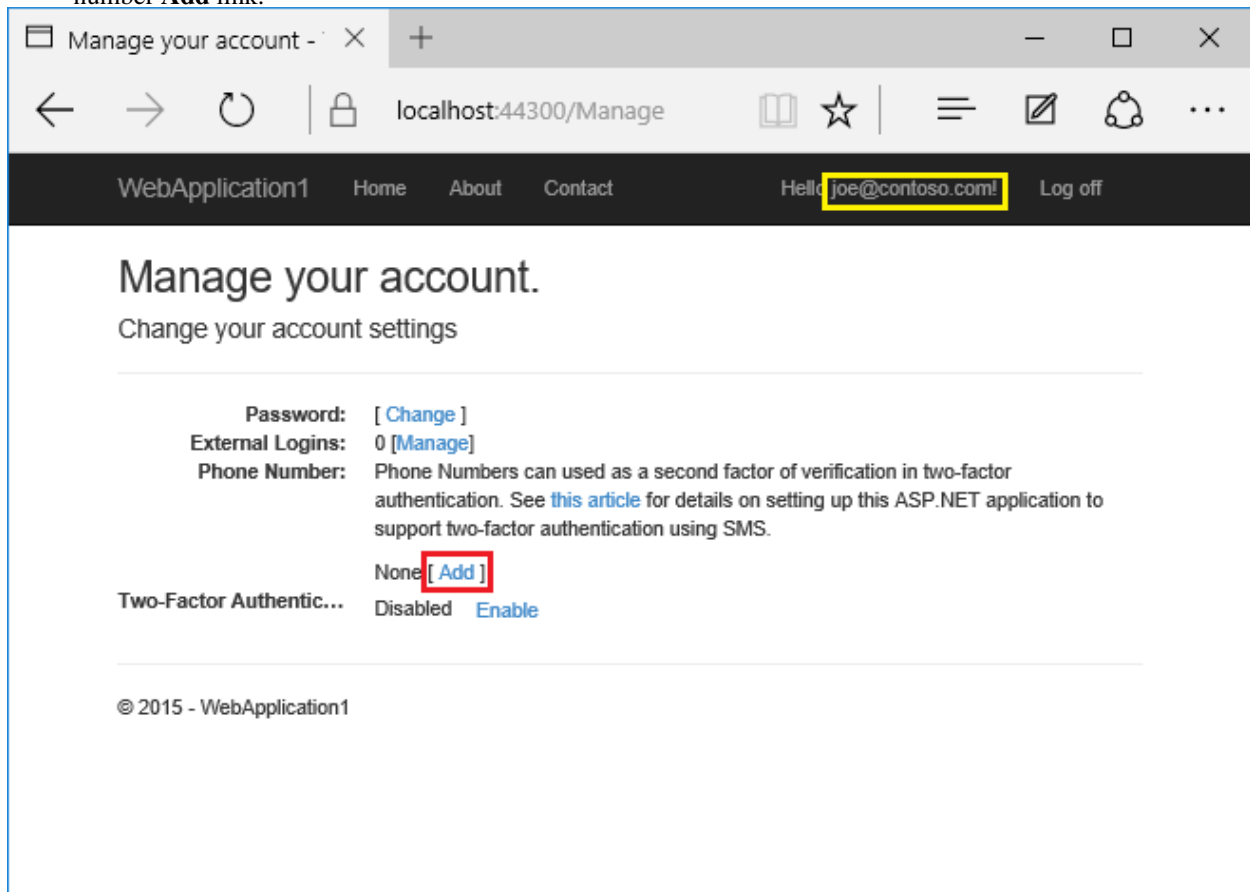
- Add a phone number that will receive the verification code, and tap **Send verification code**.

- You will get a text message with the verification code. Enter it and tap **Submit**
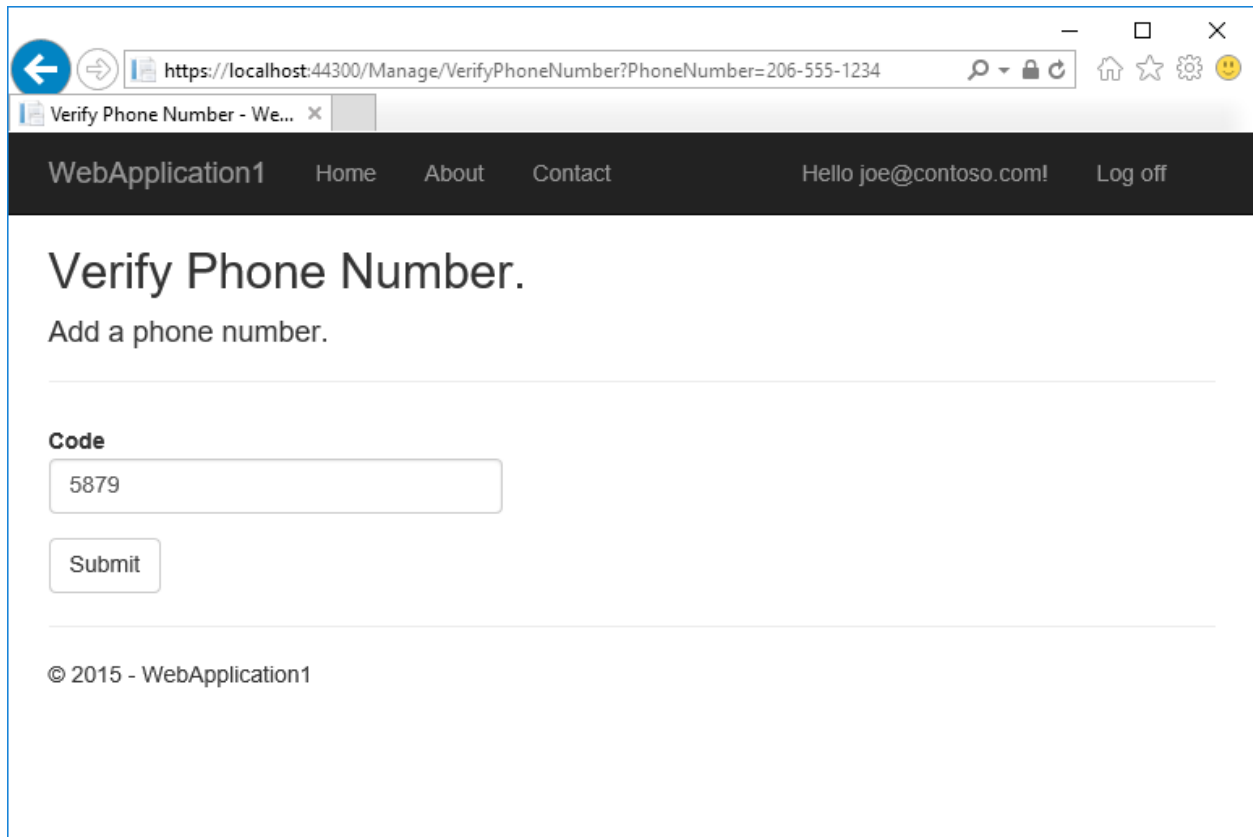
If you don't get a text message, see *Debugging Twilio*.

- The Manage view shows your phone number was added successfully.
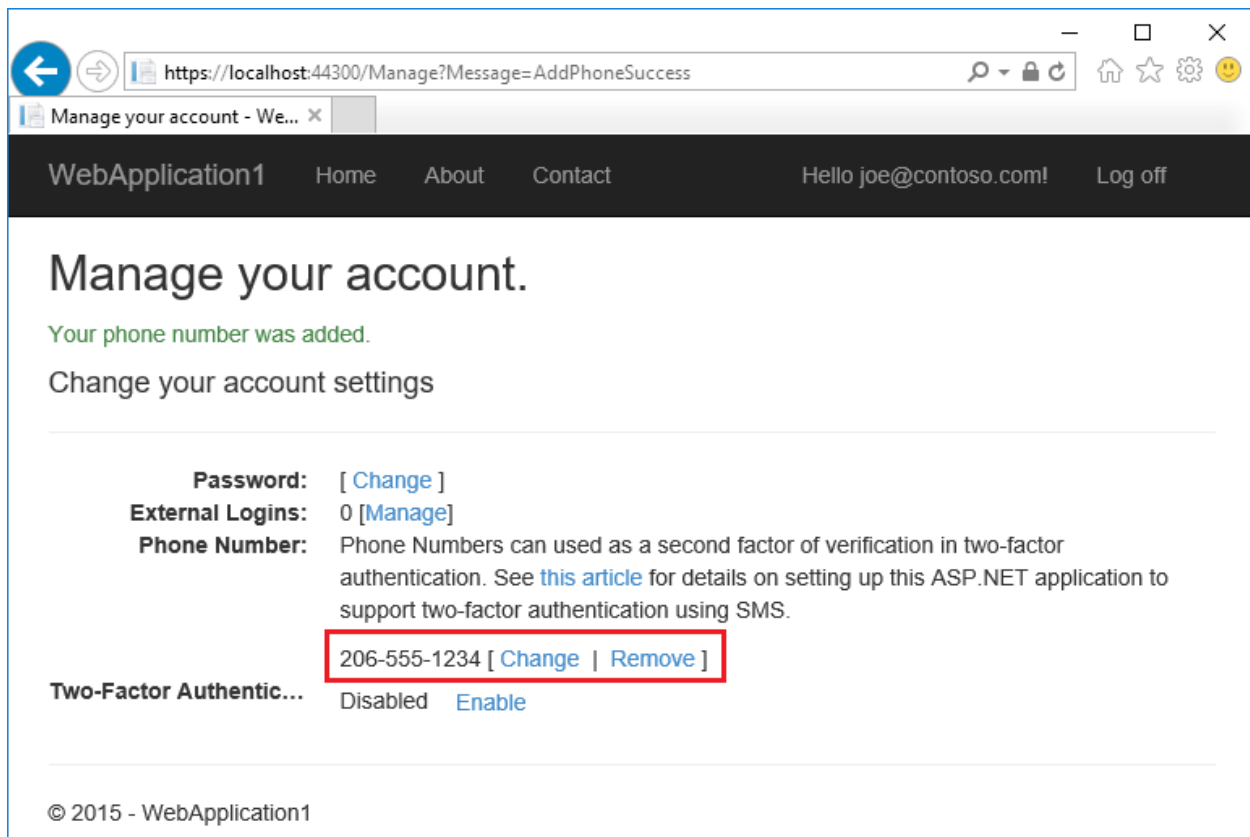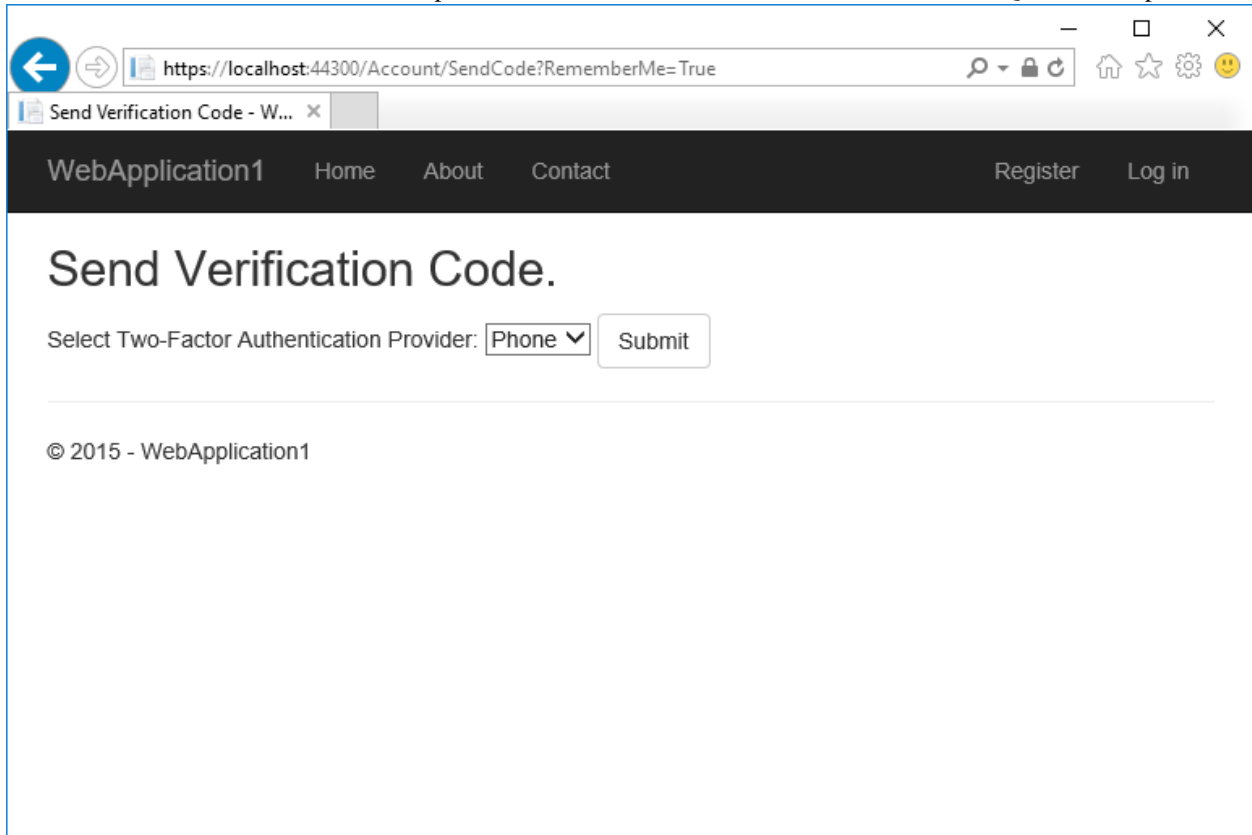
- Tap **Enable** to enable two-factor authentication.
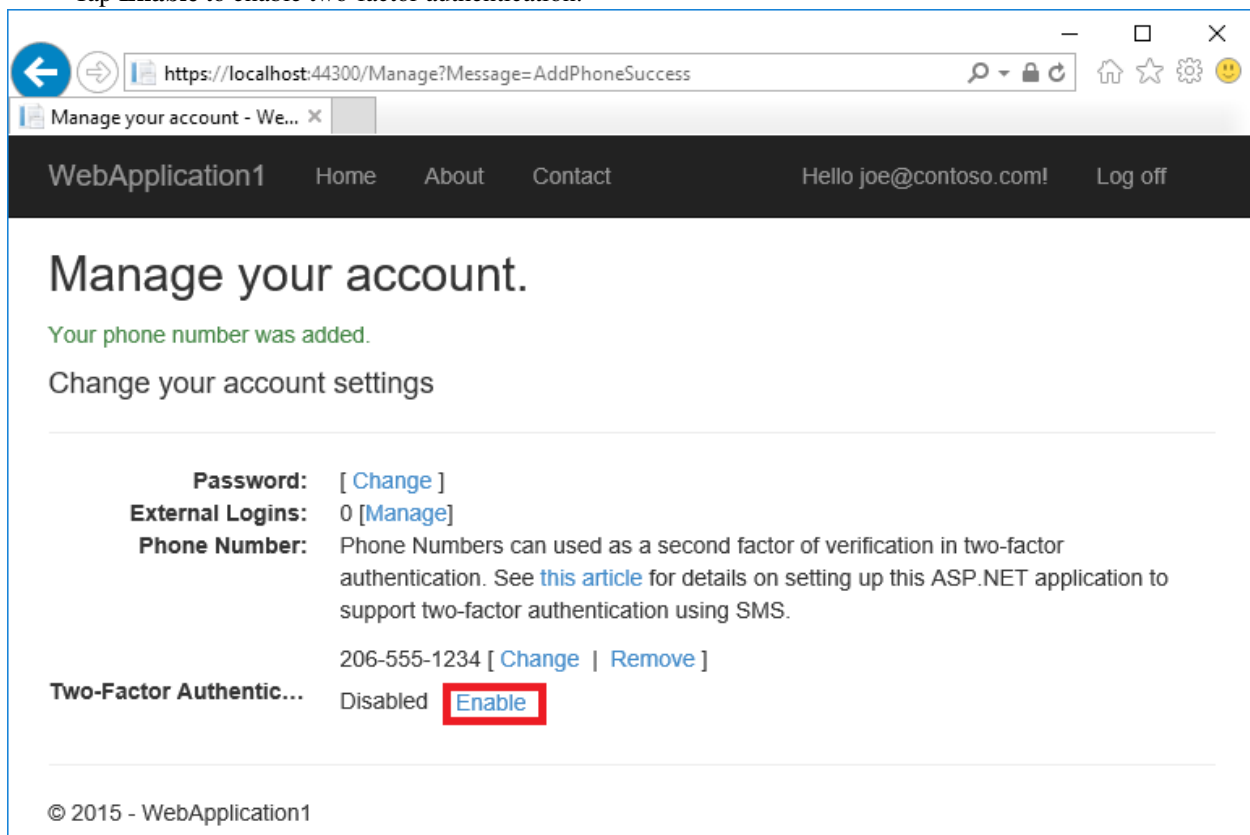
Test two-factor authentication

- Log off.

- Log in.

- The user account has enabled two-factor authentication, so you have to provide the second factor of authentication . In this tutorial you have enabled phone verification. The built in templates also allow you to set up email as the second factor. You can set up additional second factors for authentication such as QR codes. Tap **Submit**.

- Enter the code you get in the SMS message.

- Clicking on the **Remember this browser** check box will exempt you from needing to use 2FA to log on when using the same device and browser. Enabling 2FA and clicking on **Remember this browser** will provide you with strong 2FA protection from malicious users trying to access your account, as long as they don't have access to your device. You can do this on any private device you regularly use. By setting **Remember this browser**, you get the added security of 2FA from devices you don't regularly use, and you get the convenience on not having to go through 2FA on your own devices.

Account lockout for protecting against brute force attacks

We recommend you use account lockout with 2FA. Once a user logs in (through a local account or social account), each failed attempt at 2FA is stored, and if the maximum attempts (default is 5) is reached, the user is locked out for five minutes (you can set the lock out time with `DefaultAccountLockoutTimeSpan`). The following configures Account to be locked out for 10 minutes after 10 failed attempts.

```
    services.Configure<IdentityOptions>(options =>
    {
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(10);
        options.Lockout.MaxFailedAccessAttempts = 10;
    });

    // Register application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
    services.Configure<AuthMessageSMSSenderOptions>(Configuration);
}
```

Debugging Twilio

If you're able to use the Twilio API, but you don't get an SMS message, try the following:

1. Log in to the Twilio site and navigate to the **Logs > SMS & MMS Logs** page. You can verify that messages were sent and delivered.

2. Use the following code in a console application to test Twilio:

```
static void Main(string[] args)
{
  string AccountSid = "";
  string AuthToken = "";
  var twilio = new Twilio.TwilioRestClient(AccountSid, AuthToken);
  string FromPhone = "";
  string toPhone = "";
  var message = twilio.SendMessage(FromPhone, toPhone, "Twilio Test");
  Console.WriteLine(message.Sid);
}
```

Supporting Third Party Clients using OAuth 2.0

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

Using Cookie Middleware without ASP.NET Core Identity

ASP.NET Core provides cookie *middleware* which serializes a user principal into an encrypted cookie and then, on subsequent requests, validates the cookie, recreates the principal and assigns it to the `User` property on `HttpContext`. If you want to provide your own login screens and user databases you can use the cookie middleware as a standalone feature.

Adding and configuring

The first step is adding the cookie middleware to your application. First use nuget to add the `Microsoft.AspNetCore.Authentication.Cookies` package. Then add the following lines to the `Configure` method in your *Startup.cs* file;

```
app.UseCookieAuthentication(new CookieAuthenticationOptions()
{
    AuthenticationScheme = "MyCookieMiddlewareInstance",
    LoginPath = new PathString("/Account/Unauthorized/"),
    AccessDeniedPath = new PathString("/Account/Forbidden/"),
    AutomaticAuthenticate = true,
    AutomaticChallenge = true
});
```

The code snippet above configures a few options;

- AuthenticationScheme - this is a value by which the middleware is known. This is useful when there are multiple instances of middleware and you want to *limit authorization to one instance*.

- LoginPath - this is the relative path requests will be redirected to when a user attempts to access a resource but has not been authenticated.

- AccessDeniedPath - this is the relative path requests will be redirected to when a user attempts to access a resource but does not pass any *authorization policies* for that resource.

- AutomaticAuthenticate - this flag indicates that the middleware should run on every request and attempt to validate and reconstruct any serialized principal it created.

- AutomaticChallenge - this flag indicates that the middleware should redirect the browser to the `LoginPath` or `AccessDeniedPath` when authorization fails.

Other options include the ability to set the issuer for any claims the middleware creates, the name of the cookie the middleware drops, the domain for the cookie and various security properties on the cookie. By default the cookie middleware will use appropriate security options for any cookies it creates, setting HTTPONLY to avoid the cookie being accessible in client side JavaScript and limiting the cookie to HTTPS if a request has come over HTTPS.

Creating an identity cookie

To create a cookie holding your user information you must construct a ClaimsPrincipal holding the information you wish to be serialized in the cookie. Once you have a suitable *ClaimsPrincipal* inside your controller method call

```
    await HttpContext.Authentication.SignInAsync("MyCookieMiddlewareInstance", principal);
```

This will create an encrypted cookie and add it to the current response. The `AuthenticationScheme` specified during *configuration* must also be used when calling `SignInAsync`.

Under the covers the encryption used is ASP.NET's *Data Protection* system. If you are hosting on multiple machines, load balancing or using a web farm then you will need to *configure* data protection to use the same key ring and application identifier.

Signing out

To sign out the current user, and delete their cookie call the following inside your controller

```
await HttpContext.Authentication.SignOutAsync("MyCookieMiddlewareInstance");
```

Reacting to back-end changes

> : Once a principal cookie has been created it becomes the single source of identity - even if you disable a user in your back-end systems the cookie middleware has no knowledge of this and a user will continue to stay logged in as long as their cookie is valid.

The cookie authentication middleware provides a series of Events in its option class. The `ValidateAsync()` event can be used to intercept and override validation of the cookie identity.

Consider a back-end user database that may have a LastChanged column. In order to invalidate a cookie when the database changes you should first, when *creating the cookie*, add a LastChanged claim containing the current value. Then, when the database changes the LastChanged value should also be updated.

To implement an override for the `ValidateAsync()` event you must write a method with the following signature;

```
Task ValidateAsync(CookieValidatePrincipalContext context);
```

ASP.NET Core Identity implements this check as part of its SecurityStampValidator. A simple example would look something like as follows;

```
public static class LastChangedValidator
{
    public static async Task ValidateAsync(CookieValidatePrincipalContext context)
    {
        // Pull database from registered DI services.
        var userRepository = context.HttpContext.RequestServices.GetRequiredService<IUserRepository>
        var userPrincipal = context.Principal;

        // Look for the last changed claim.
        string lastChanged;
        lastChanged = (from c in userPrincipal.Claims
                       where c.Type == "LastUpdated"
                       select c.Value).FirstOrDefault();

        if (string.IsNullOrEmpty(lastChanged) ||
            !userRepository.ValidateLastChanged(userPrincipal, lastChanged))
        {
            context.RejectPrincipal();
            await context.HttpContext.Authentication.SignOutAsync("MyCookieMiddlewareInstance");
        }
    }
}
```

This would then be wired up during cookie middleware configuration

```
app.UseCookieAuthentication(options =>
{
    options.Events = new CookieAuthenticationEvents
    {
        // Set other options
        OnValidatePrincipal = LastChangedValidator.ValidateAsync
    };
});
```

If you want to non-destructively update the user principal, for example, their name might have been updated, a decision which doesn't affect security in any way you can call `context.ReplacePrincipal()` and set the `context.ShouldRenew` flag to `true`.

Controlling cookie options

The `CookieAuthenticationOptions` class comes with various configuration options to enable you to fine tune the cookies created.

- **ClaimsIssuer** - the issuer to be used for the Issuer property on any claims created by the middleware.

- **CookieDomain** - the domain name the cookie will be served to. By default this is the host name the request was sent to. The browser will only serve the cookie to a matching host name. You may wish to adjust this to have cookies available to any host in your domain. For example setting the cookie domain to `.contoso.com` will make it available to `contoso.com`, `www.contoso.com`, `staging.www.contoso.com` etc.

- **CookieHttpOnly** - a flag indicating if the cookie should only be accessible to servers. This defaults to `true`. Changing this value may open your application to cookie theft should your application have a Cross Site Scripting bug.

- **CookiePath** - this can be used to isolate applications running on the same host name. If you have an app running in `/app1` and want to limit the cookies issued to just be sent to that application then you should set the `CookiePath` property to `/app1`. The cookie will now only be available to requests to `/app1` or anything underneath it.

- **CookieSecure** - a flag indicating if the cookie created should be limited to HTTPS, HTTP or HTTPS, or the same protocol as the request. This defaults to `SameAsRequest`.

- **ExpireTimeSpan** - the `TimeSpan` after which the cookie will expire. This is added to the current date and time to create the expiry date for the cookie.

- **SlidingExpiration** - a flag indicating if the cookie expiration date will be reset when the more than half of the `ExpireTimeSpan` interval has passed. The new expiry date will be moved forward to be the current date plus the `ExpireTimespan`. An *absolute expiry time* can be set by using the `AuthenticationProperties` class when calling `SignInAsync`. An absolute expiry can improve the security of your application by limiting the amount of time for which the authentication cookie is valid.

Persistent cookies and absolute expiry times

You may want to make the cookie expire be remembered over browser sessions. You may also want an absolute expiry to the identity and the cookie transporting it. You can do these things by using the `AuthenticationProperties` parameter on the `HttpContext.Authentication.SignInAsync` method called when *signing in an identity and creating the cookie*. The `AuthenticationProperties` class is in the `Microsoft.AspNetCore.Http.Authentication` namespace.

For example;

```
await HttpContext.Authentication.SignInAsync(
    "MyCookieMiddlewareInstance",
    principal,
    new AuthenticationProperties
    {
        IsPersistent = true
    });
```

This code snippet will create an identity and corresponding cookie which will be survive through browser closures.Any sliding expiration settings previously configured via *cookie options* will still be honored, if the cookie expires whilst the browser is closed the browser will clear it once it is restarted.

```
await HttpContext.Authentication.SignInAsync(
    "MyCookieMiddlewareInstance",
    principal,
    new AuthenticationProperties
    {
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
    });
```

This code snippet will create an identity and corresponding cookie which will be last for 20 minutes. This ignores any sliding expiration settings previously configured via *cookie options*.

The ExpiresUtc and IsPersistent properties are mutually exclusive.

Azure Active Directory

1.12.2 Authorization

Introduction

Authorization refers to the process that determines what a user is able to do. For example user Adam may be able to create a document library, add documents, edit documents and delete them. User Bob may only be authorized to read documents in a single library.

Authorization is orthogonal and independent from authentication, which is the process of ascertaining who a user is. Authentication may create one or more identities for the current user.

Authorization Types

In ASP.NET Core authorization now provides simple declarative *role* and a *richer policy based* model where authorization is expressed in requirements and handlers evaluate a users claims against requirements. Imperative checks can be based on simple policies or polices which evaluate both the user identity and properties of the resource that the user is attempting to access.

Namespaces

Authorization components, including the `AuthorizeAttribute` and `AllowAnonymousAttribute` attributes are found in the `Microsoft.AspNetCore.Authorization` namespace.

Simple Authorization

Authorization in MVC is controlled through the `AuthorizeAttribute` attribute and its various parameters. At its simplest applying the `AuthorizeAttribute` attribute to a controller or action limits access to the controller or action to any authenticated user.

For example, the following code limits access to the `AccountController` to any authenticated user.

```
[Authorize]
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

If you want to apply authorization to an action rather than the controller simply apply the `AuthorizeAttribute` attribute to the action itself;

```
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }
```

```
    [Authorize]
    public ActionResult Logout()
    {
    }
}
```

Now only authenticated users can access the logout function.

You can also use the `AllowAnonymousAttribute` attribute to allow access by non-authenticated users to individual actions; for example

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

This would allow only authenticated users to the `AccountController`, except for the `Login` action, which is accessible by everyone, regardless of their authenticated or unauthenticated / anonymous status.

> : `[AllowAnonymous]` bypasses all authorization statements. If you apply combine `[AllowAnonymous]` and any `[Authorize]` attribute then the Authorize attributes will always be ignored. For example if you apply `[AllowAnonymous]` at the controller level any `[Authorize]` attributes on the same controller, or on any action within it will be ignored.

Role based Authorization

When an identity is created it may belong to one or more roles, for example Tracy may belong to the Administrator and User roles whilst Scott may only belong to the user role. How these roles are created and managed depends on the backing store of the authorization process. Roles are exposed to the developer through the IsInRole property on the ClaimsPrincipal class.

Adding role checks

Role based authorization checks are declarative - the developer embeds them within their code, against a controller or an action within a controller, specifying roles which the current user must be a member of to access the requested resource.

For example the following code would limit access to any actions on the `AdministrationController` to users who are a member of the `Administrator` group.

```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{
}
```

You can specify multiple roles as a comma separated list;

```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
}
```

This controller would be only accessible by users who are members of the `HRManager` role or the `Finance` role.

If you apply multiple attributes then an accessing user must be a member of all the roles specified; the following sample requires that a user must be a member of both the `PowerUser` and `ControlPanelUser` role.

```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
}
```

You can further limit access by applying additional role authorization attributes at the action level;

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

In the previous code snippet members of the `Administrator` role or the `PowerUser` role can access the controller and the `SetTime` action, but only members of the `Administrator` role can access the `ShutDown` action.

You can also lock down a controller but allow anonymous, unauthenticated access to individual actions.

```
[Authorize]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [AllowAnonymous]
    public ActionResult Login()
    {
    }
}
```

Policy based role checks

Role requirements can also be expressed using the new Policy syntax, where a developer registers a policy at startup as part of the Authorization service configuration. This normally takes part in `ConfigureServices()` in your *Startup.cs* file.

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdministratorRole", policy => policy.RequireRole("Administrator"));
    }
}
```

Policies are applied using the `Policy` property on the `AuthorizeAttribute` attribute;

```
[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown()
{
    return View();
}
```

If you want to specify multiple allowed roles in a requirement then you can specify them as parameters to the `RequireRole` method;

```
options.AddPolicy("ElevatedRights", policy =>
                  policy.RequireRole("Administrator", "PowerUser", "BackupAdministrator"));
```

This example authorizes users who belong to the `Administrator`, `PowerUser` or `BackupAdministrator` roles.

Claims-Based Authorization

When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is name value pair that represents what the subject is, not what the subject can do. For example you may have a Drivers License, issued by a local driving license authority. Your driver's license has your date of birth on it. In this case the claim name would be `DateOfBirth`, the claim value would be your date of birth, for example `8th June 1970` and the issuer would be the driving license authority. Claims based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value. For example if you want access to a night club the authorization process might be:

The door security officer would evaluate the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.

An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

Adding claims checks

Claim based authorization checks are declarative - the developer embeds them within their code, against a controller or an action within a controller, specifying claims which the current user must possess, and optionally the value the claim must hold to access the requested resource. Claims requirements are policy based, the developer must build and register a policy expressing the claims requirements.

The simplest type of claim policy looks for the presence of a claim and does not check the value.

First you need to build and register the policy. This takes place as part of the Authorization service configuration, which normally takes part in `ConfigureServices()` in your *Startup.cs* file.

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("EmployeeNumber"));
    });
}
```

In this case the `EmployeeOnly` policy checks for the presence of an `EmployeeNumber` claim on the current identity.

You then apply the policy using the `Policy` property on the `AuthorizeAttribute` attribute to specify the policy name;

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    return View();
}
```

The `AuthorizeAttribute` attribute can be applied to an entire controller, in this instance only identities matching the policy will be allowed access to any Action on the controller.

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public ActionResult VacationBalance()
    {
    }
}
```

If you have a controller that is protected by the `AuthorizeAttribute` attribute, but want to allow anonymous access to particular actions you apply the `AllowAnonymousAttribute` attribute;

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public ActionResult VacationBalance()
    {
    }

    [AllowAnonymous]
    public ActionResult VacationPolicy()
    {
    }
}
```

Most claims come with a value. You can specify a list of allowed values when creating the policy. The following example would only succeed for employees whose employee number was 1, 2, 3, 4 or 5.

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

    services.AddAuthorization(options =>
```

```
    {
        options.AddPolicy("Founders", policy =>
                            policy.RequireClaim("EmployeeNumber", "1", "2", "3", "4", "5"));
    }
}
```

Multiple Policy Evaluation

If you apply multiple policies to a controller or action then all policies must pass before access is granted. For example;

```
[Authorize(Policy = "EmployeeOnly")]
public class SalaryController : Controller
{
    public ActionResult Payslip()
    {
    }

    [Authorize(Policy = "HumanResources")]
    public ActionResult UpdateSalary()
    {
    }
}
```

In the above example any identity which fulfills the `EmployeeOnly` policy can access the `Payslip` action as that policy is enforced on the controller. However in order to call the `UpdateSalary` action the identity must fulfill *both* the `EmployeeOnly` policy and the `HumanResources` policy.

If you want more complicated policies, such as taking a date of birth claim, calculating an age from it then checking the age is 21 or older then you need to write *custom policy handlers*.

Custom Policy-Based Authorization

Underneath the covers the *role authorization* and *claims authorization* make use of a requirement, a handler for the requirement and a pre-configured policy. These building blocks allow you to express authorization evaluations in code, allowing for a richer, reusable, and easily testable authorization structure.

An authorization policy is made up of one or more requirements and registered at application startup as part of the Authorization service configuration, in `ConfigureServices` in the *Startup.cs* file.

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Over21",
                            policy => policy.Requirements.Add(new MinimumAgeRequirement(21)));
    }
});
```

Here you can see an "Over21" policy is created with a single requirement, that of a minimum age, which is passed as a parameter to the requirement.

Policies are applied using the `Authorize` attribute by specifying the policy name, for example;

```
[Authorize(Policy="Over21")]
public class AlcoholPurchaseRequirementsController : Controller
{
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

Requirements

An authorization requirement is a collection of data parameters that a policy can use to evaluate the current user
principal. In our Minimum Age policy the requirement we have a single parameter, the minimum age. A requirement
must implement IAuthorizationRequirement. This is an empty, marker interface. A parameterized minimum
age requirement might be implemented as follows;

```
public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public MinimumAgeRequirement(int age)
    {
        MinimumAge = age;
    }

    protected int MinimumAge { get; set; }
}
```

A requirement doesn't need to have data or properties.

Authorization Handlers

An authorization handler is responsible for the evaluation of any properties of a requirement. The authorization
handler must evaluate them against a provided AuthorizationContext to decide if authorization is allowed.
A requirement can have *multiple handlers*. Handlers must inherit AuthorizationHandler<T> where T is the
requirement it handles. The minimum age handler might look like this:

```
public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationContext context, MinimumAgeRequiremen
    {
        if (!context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth &&
                                   c.Issuer == "http://contoso.com"))
        {
            return;
        }

        var dateOfBirth = Convert.ToDateTime(context.User.FindFirst(
            c => c.Type == ClaimTypes.DateOfBirth && c.Issuer == "http://contoso.com").Value);

        int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;
        if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))
        {
```

```
            calculatedAge--;
        }

        if (calculatedAge >= requirement.MinimumAge)
        {
            context.Succeed(requirement);
        }
        return Task.FromResult(0);
    }
}
```

In the code above we first look to see if the current user principal has a date of birth claim which has been issued by an Issuer we know and trust. If the claim is missing we can't authorize so we return. If we have a claim, we figure out how old the user is, and if they meet the minimum age passed in by the requirement then authorization has been successful. Once authorization is successful we call `context.Succeed()` passing in the requirement that has been successful as a parameter. Handlers must be registered in the services collection during configuration, for example;

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Over21",
                        policy => policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });

    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
}
```

Each handler is added to the services collection by using `services.AddSingleton<IAuthorizationHandler, YourHandlerClass>();` passing in your handler class.

What should a handler return?

You can see in our *handler example* that the `Handle()` method has no return value, so how do we indicate success or failure?

- A handler indicates success by calling `context.Succeed(IAuthorizationRequirement requirement)`, passing the requirement that has been successfully validated.

- A handler does not need to handle failures generally, as other handlers for the same requirement may succeed.

- To guarantee failure even if other handlers for a requirement succeed, call `context.Fail`.

Regardless of what you call inside your handler all handlers for a requirement will be called when a policy requires the requirement. This allows requirements to have side effects, such as logging, which will always take place even if `context.Fail()` has been called in another handler.

Why would I want multiple handlers for a requirement?

In cases where you want evaluation to be on an **OR** basis you implement multiple handlers for a single requirement. For example, Microsoft has doors which only open with key cards. If you leave your key card at home the receptionist prints a temporary sticker and opens the door for you. In this scenario you'd have a single requirement, *EnterBuilding*, but multiple handlers, each one examining a single requirement.

```csharp
public class EnterBuildingRequirement : IAuthorizationRequirement
{
}

public class BadgeEntryHandler : AuthorizationHandler<EnterBuildingRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationContext context, EnterBuildingRequire
    {
        if (context.User.HasClaim(c => c.Type == ClaimTypes.BadgeId &&
                                       c.Issuer == "http://microsoftsecurity"))
        {
            context.Succeed(requirement);
            return Task.FromResult(0);
        }
    }
}

public class HasTemporaryStickerHandler : AuthorizationHandler<EnterBuildingRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationContext context, EnterBuildingRequire
    {
        if (context.User.HasClaim(c => c.Type == ClaimTypes.TemporaryBadgeId &&
                                       c.Issuer == "https://microsoftsecurity"))
        {
            // We'd also check the expiration date on the sticker.
            context.Succeed(requirement);
            return Task.FromResult(0);
        }
    }
}
```

Now, assuming both handlers are *registered* when a policy evaluates the EnterBuildingRequirement if either handler succeeds the policy evaluation will succeed.

Using a func to fufill a policy

There may be occasions where fufilling a policy is simple to express in code. It is possible to simply supply a Func<AuthorizationHandlerContext, bool> when configuring your policy with the RequireAssertion policy builder.

For example the previous BadgeEntryHandler could be rewritten as follows;

```csharp
services.AddAuthorization(options =>
    {
        options.AddPolicy("BadgeEntry",
                       policy => policy.RequireAssertion(context =>
                            context.User.HasClaim(c =>
                                (c.Type == ClaimTypes.BadgeId ||
                                 c.Type == ClaimTypes.TemporaryBadgeId)
                                && c.Issuer == "https://microsoftsecurity"));
                       }));
    }
 }
```

Accessing MVC Request Context In Handlers

The `Handle` method you must implement in an authorization handler has two parameters, an `AuthorizationContext` and the `Requirement` you are handling. Frameworks such as MVC or Jabbr are free to add any object to the `Resource` property on the `AuthorizationContext` to pass through extra information.

For example MVC passes an instance of `Microsoft.AspNetCore.Mvc.Filters.AuthorizationFilterContext` in the resource property which is used to access HttpContext, RouteData and everything else MVC provides.

The use of the `Resource` property is framework specific. Using information in the `Resource` property will limit your authorization policies to particular frameworks. You should cast the `Resource` property using the `as` keyword, and then check the cast has succeed to ensure your code doesn't crash with `InvalidCastExceptions` when run on other frameworks;

```
var mvcContext = context.Resource as Microsoft.AspNetCore.Mvc.Filters.AuthorizationFilterContext;

if (mvcContext != null)
{
    // Examine MVC specific things like routing data.
}
```

Dependency Injection in Requirement Handlers

As *handlers must be registered* in the service collection they support *dependency injection*. If, for example, you had a repository of rules you want to evaluate inside a handler and that repository is registered in the service collection authorization will resolve and inject that into your constructor.

For example, if you wanted to use ASP.NET's logging infrastructure you would to inject `ILoggerFactory` into your handler. Such a handler might look like this;

```
public class LoggingAuthorizationHandler : AuthorizationHandler<MyRequirement>
{
    ILogger _logger;

    public LoggingAuthorizationHandler(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger(this.GetType().FullName);
    }

    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, MyRequirement
    {
        _logger.LogInformation("Inside my handler");
        // Check if the requirement is fulfilled.
        return Task.CompletedTask;
    }
}
```

Then you register handlers with `services.AddSingleton()`, for example

```
services.AddSingleton<IAuthorizationHandler, LoggingAuthorizationHandler>();
```

An instance of the handler will be created when your application starts, and DI will inject the registered `ILoggerFactory` into your constructor.

Resource Based Authorization

Often authorization depends upon the resource being accessed. For example a document may have an author property. Only the document author would be allowed to update it, so the resource must be loaded from the document repository before an authorization evaluation can be made. This cannot be done with an Authorize attribute, as attribute evaluation takes place before data binding and before your own code to load a resource runs inside an action. Instead of declarative authorization, the attribute method, we must use imperative authorization, where a developer calls an authorize function within his own code.

Authorizing within your code

Authorization is implemented as a service, `IAuthorizationService`, registered in the service collection and available via *dependency injection* for Controllers to access.

```
public class DocumentController : Controller
{
    IAuthorizationService _authorizationService;

    public DocumentController(IAuthorizationService authorizationService)
    {
        _authorizationService = authorizationService;
    }
}
```

`IAuthorizationService` has two methods, one where you pass the resource and the policy name and the other where you pass the resource and a list of requirements to evaluate.

```
Task<bool> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          IEnumerable<IAuthorizationRequirement> requirements);
Task<bool> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          string policyName);
```

To call the service load your resource within your action then call the `AuthorizeAsync` overload you require. For example

```
public async Task<IActionResult> Edit(Guid documentId)
{
    Document document = documentRepository.Find(documentId);

    if (document == null)
    {
        return new HttpNotFoundResult();
    }

    if (await authorizationService.AuthorizeAsync(User, document, "EditPolicy"))
    {
        return View(document);
    }
    else
    {
        return new ChallengeResult();
    }
}
```

Writing a resource based handler

Writing a handler for resource based authorization is not that much different to *writing a plain requirements handler*. You create a requirement, and then implement a handler for the requirement, specifying the requirement as before and also the resource type. For example, a handler which might accept a Document resource would look as follows;

```csharp
public class DocumentAuthorizationHandler : AuthorizationHandler<MyRequirement, Document>
{
    public override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                MyRequirement requirement,
                                                Document resource)
    {
        // Validate the requirement against the resource and identity.

        return Task.CompletedTask;
    }
}
```

Don't forget you also need to register your handler in the `ConfigureServices` method;

```csharp
services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationHandler>();
```

Operational Requirements If you are making decisions based on operations such as read, write, update and delete, you can use the `OperationAuthorizationRequirement` class in the `Microsoft.AspNetCore.Authorization.Infrastructure` namespace. This prebuilt requirement class enables you to write a single handler which has a parameterized operation name, rather than create individual classes for each operation. To use it provide some operation names:

```csharp
public static class Operations
{
    public static OperationAuthorizationRequirement Create =
        new OperationAuthorizationRequirement { Name = "Create" };
    public static OperationAuthorizationRequirement Read =
        new OperationAuthorizationRequirement   { Name = "Read" };
    public static OperationAuthorizationRequirement Update =
        new OperationAuthorizationRequirement { Name = "Update" };
    public static OperationAuthorizationRequirement Delete =
        new OperationAuthorizationRequirement { Name = "Delete" };
}
```

Your handler could then be implemented as follows, using a hypothetical `Document` class as the resource;

```csharp
public class DocumentAuthorizationHandler :
    AuthorizationHandler<OperationAuthorizationRequirement, Document>
{
    public override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                OperationAuthorizationRequirement requirement,
                                                Document resource)
    {
        // Validate the operation using the resource, the identity and
        // the Name property value from the requirement.

        return Task.CompletedTask;
    }
}
```

You can see the handler works on `OperationAuthorizationRequirement`. The code inside the handler must take the Name property of the supplied requirement into account when making its evaluations.

To call an operational resource handler you need to specify the operation when calling `AuthorizeAsync` in your action. For example

```
if (await authorizationService.AuthorizeAsync(User, document, Operations.Read))
{
    return View(document);
}
else
{
    return new ChallengeResult();
}
```

This example checks if the User is able to perform the Read operation for the current `document` instance. If authorization succeeds the view for the document will be returned. If authorization fails returning `ChallengeResult` will inform any authentication middleware authorization has failed and the middleware can take the appropriate response, for example returning a 401 or 403 status code, or redirecting the user to a login page for interactive browser clients.

View Based Authorization

Often a developer will want to show, hide or otherwise modify a UI based on the current user identity. You can access the authorization service within MVC views via *dependency injection*. To inject the authorization service into a Razor view use the `@inject` directive, for example `@inject IAuthorizationService AuthorizationService`. If you want the authorization service in every view then place the `@inject` directive into the `_ViewImports.cshtml` file in the `Views` directory.

Once you have injected the authorization service you use it by calling the `AuthorizeAsync` method in exactly the same way as you would check during *resource based authorization*.

```
@if (await AuthorizationService.AuthorizeAsync(User, "PolicyName"))
{
    <p>This paragraph is displayed because you fulfilled PolicyName.</p>
}
```

In some cases the resource will be your view model, and you can call `AuthorizeAsync` in exactly the same way as you would check during *resource based authorization*;

```
@if (await AuthorizationService.AuthorizeAsync(User, Model, Operations.Edit))
{
    <p><a class="btn btn-default" role="button"
        href="@Url.Action("Edit", "Document", new { id = Model.Id })">Edit</a></p>
}
```

Here you can see the model is passed as the resource authorization should take into consideration.

> **:** Do not rely on showing or hiding parts of your UI as your only authorization method. Hiding a UI element does not mean a user cannot access it. You must also authorize the user within your controller code.

Limiting identity by scheme

In some scenarios, such as Single Page Applications it is possible to end up with multiple authentication methods. For example, your application may use cookie-based authentication to log in and bearer authentication for JavaScript requests. In some cases you may have multiple instances of an authentication middleware. For example, two cookie middlewares where one contains a basic identity and one is created when a multi-factor authentication has triggered because the user requested an operation that requires extra security.

Authentication schemes are named when authentication middleware is configured during authentication, for example

```
app.UseCookieAuthentication(new CookieAuthenticationOptions()
{
    AuthenticationScheme = "Cookie",
    LoginPath = new PathString("/Account/Unauthorized/"),
    AccessDeniedPath = new PathString("/Account/Forbidden/"),
    AutomaticAuthenticate = false
});

app.UseBearerAuthentication(options =>
{
    options.AuthenticationScheme = "Bearer";
    options.AutomaticAuthenticate = false;
});
```

In this configuration two authentication middlewares have been added, one for cookies and one for bearer.

> **:** When adding multiple authentication middleware you should ensure that no middleware is configured to run automatically. You do this by setting the `AutomaticAuthenticate` options property to false. If you fail to do this filtering by scheme will not work.

Selecting the scheme with the Authorize attribute

As no authentication middleware is configured to automatically run and create an identity you must, at the point of authorization choose which middleware will be used. The simplest way to select the middleware you wish to authorize with is to use the `ActiveAuthenticationSchemes` property. This property accepts a comma delimited list of Authentication Schemes to use. For example;

```
[Authorize(ActiveAuthenticationSchemes = "Cookie,Bearer")]
public class MixedController : Controller
```

In the example above both the cookie and bearer middlewares will run and have a chance to create and append an identity for the current user. By specifying a single scheme only the specified middleware will run;

```
[Authorize(ActiveAuthenticationSchemes = "Bearer")]
```

In this case only the middleware with the Bearer scheme would run, and any cookie based identities would be ignored.

Selecting the scheme with policies

If you prefer to specify the desired schemes in *policy* you can set the `AuthenticationSchemes` collection when adding your policy.

```
options.AddPolicy("Over18", policy =>
{
    policy.AuthenticationSchemes.Add("Bearer");
    policy.RequireAuthenticatedUser();
    policy.Requirements.Add(new Over18Requirement());
});
```

In this example the Over18 policy will only run against the identity created by the `Bearer` middleware.

Authorization Filters

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.12.3 Data Protection

Introduction to Data Protection

Web applications often need to store security-sensitive data. Windows provides DPAPI for desktop applications but this is unsuitable for web applications. The ASP.NET Core data protection stack provide a simple, easy to use cryptographic API a developer can use to protect data, including key management and rotation.

The ASP.NET Core data protection stack is designed to serve as the long-term replacement for the <machineKey> element in ASP.NET 1.x - 4.x. It was designed to address many of the shortcomings of the old cryptographic stack while providing an out-of-the-box solution for the majority of use cases modern applications are likely to encounter.

Problem statement

The overall problem statement can be succinctly stated in a single sentence: I need to persist trusted information for later retrieval, but I do not trust the persistence mechanism. In web terms, this might be written as "I need to round-trip trusted state via an untrusted client."

The canonical example of this is an authentication cookie or bearer token. The server generates an "I am Groot and have xyz permissions" token and hands it to the client. At some future date the client will present that token back to the server, but the server needs some kind of assurance that the client hasn't forged the token. Thus the first requirement: authenticity (a.k.a. integrity, tamper-proofing).

Since the persisted state is trusted by the server, we anticipate that this state might contain information that is specific to the operating environment. This could be in the form of a file path, a permission, a handle or other indirect reference, or some other piece of server-specific data. Such information should generally not be disclosed to an untrusted client. Thus the second requirement: confidentiality.

Finally, since modern applications are componentized, what we've seen is that individual components will want to take advantage of this system without regard to other components in the system. For instance, if a bearer token component is using this stack, it should operate without interference from an anti-CSRF mechanism that might also be using the same stack. Thus the final requirement: isolation.

We can provide further constraints in order to narrow the scope of our requirements. We assume that all services operating within the cryptosystem are equally trusted and that the data does not need to be generated or consumed outside of the services under our direct control. Furthermore, we require that operations are as fast as possible since each request to the web service might go through the cryptosystem one or more times. This makes symmetric cryptography ideal for our scenario, and we can discount asymmetric cryptography until such a time that it is needed.

Design philosophy

We started by identifying problems with the existing stack. Once we had that, we surveyed the landscape of existing solutions and concluded that no existing solution quite had the capabilities we sought. We then engineered a solution based on several guiding principles.

- The system should offer simplicity of configuration. Ideally the system would be zero-configuration and developers could hit the ground running. In situations where developers need to configure a specific aspect (such as the key repository), consideration should be given to making those specific configurations simple.

- Offer a simple consumer-facing API. The APIs should be easy to use correctly and difficult to use incorrectly.

- Developers should not learn key management principles. The system should handle algorithm selection and key lifetime on the developer's behalf. Ideally the developer should never even have access to the raw key material.

- Keys should be protected at rest when possible. The system should figure out an appropriate default protection mechanism and apply it automatically.

With these principles in mind we developed a simple, *easy to use* data protection stack.

The ASP.NET Core data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like Windows CNG DPAPI and Azure Rights Management are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there is nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data.

Audience

The data protection system is divided into five main packages. Various aspects of these APIs target three main audiences;

1. The *Consumer APIs Overview* target application and framework developers.

   "I don't want to learn about how the stack operates or about how it is configured. I simply want to perform some operation in as simple a manner as possible with high probability of using the APIs successfully."

2. The *configuration APIs* target application developers and system administrators.

   "I need to tell the data protection system that my environment requires non-default paths or settings."

3. The extensibility APIs target developers in charge of implementing custom policy. Usage of these APIs would be limited to rare situations and experienced, security aware developers.

   "I need to replace an entire component within the system because I have truly unique behavioral requirements. I am willing to learn uncommonly-used parts of the API surface in order to build a plugin that fulfills my requirements."

Package Layout

The data protection stack consists of five packages.

- Microsoft.AspNetCore.DataProtection.Abstractions contains the basic IDataProtectionProvider and IDataProtector interfaces. It also contains useful extension methods that can assist working with these types (e.g., overloads of IDataProtector.Protect). See the consumer interfaces section for more information. If somebody else is responsible for instantiating the data protection system and you are simply consuming the APIs, you'll want to reference Microsoft.AspNetCore.DataProtection.Abstractions.

- Microsoft.AspNetCore.DataProtection contains the core implementation of the data protection system, including the core cryptographic operations, key management, configuration, and extensibility. If you're responsible for instantiating the data protection system (e.g., adding it to an IServiceCollection) or modifying or extending its behavior, you'll want to reference Microsoft.AspNetCore.DataProtection.

- Microsoft.AspNetCore.DataProtection.Extensions contains additional APIs which developers might find useful but which don't belong in the core package. For instance, this package contains a simple "instantiate the system pointing at a specific key storage directory with no dependency injection setup" API (more info). It also contains extension methods for limiting the lifetime of protected payloads (more info).

- Microsoft.AspNetCore.DataProtection.SystemWeb can be installed into an existing ASP.NET 4.x application to redirect its <machineKey> operations to instead use the new data protection stack. See *compatibility* for more information.

- Microsoft.AspNetCore.Cryptography.KeyDerivation provides an implementation of the PBKDF2 password hashing routine and can be used by systems which need to handle user passwords securely. See *Password Hashing* for more information.

Getting Started with the Data Protection APIs

At its simplest protecting data consists of the following steps:

1. Create a data protector from a data protection provider.

2. Call the Protect method with the data you want to protect.

3. Call the Unprotect method with the data you want to turn back into plain text.

Most frameworks such as ASP.NET or SignalR already configure the data protection system and add it to a service container you access via dependency injection. The following sample demonstrates configuring a service container for dependency injection and registering the data protection stack, receiving the data protection provider via DI, creating a protector and protecting then unprotecting data

```
1   using System;
2   using Microsoft.AspNetCore.DataProtection;
3   using Microsoft.Extensions.DependencyInjection;
4
5   public class Program
6   {
7       public static void Main(string[] args)
8       {
9           // add data protection services
10          var serviceCollection = new ServiceCollection();
11          serviceCollection.AddDataProtection();
12          var services = serviceCollection.BuildServiceProvider();
13
14          // create an instance of MyClass using the service provider
15          var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
16          instance.RunSample();
17      }
18
19      public class MyClass
```

```
20      {
21          IDataProtector _protector;
22
23          // the 'provider' parameter is provided by DI
24          public MyClass(IDataProtectionProvider provider)
25          {
26              _protector = provider.CreateProtector("Contoso.MyClass.v1");
27          }
28
29          public void RunSample()
30          {
31              Console.Write("Enter input: ");
32              string input = Console.ReadLine();
33
34              // protect the payload
35              string protectedPayload = _protector.Protect(input);
36              Console.WriteLine($"Protect returned: {protectedPayload}");
37
38              // unprotect the payload
39              string unprotectedPayload = _protector.Unprotect(protectedPayload);
40              Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
41          }
42      }
43  }
44
45  /*
46   * SAMPLE OUTPUT
47   *
48   * Enter input: Hello world!
49   * Protect returned: CfDJ8ICcgQwZZhlAlTZT...OdfH66i1PnGmpCR5e441xQ
50   * Unprotect returned: Hello world!
51   */
```

When you create a protector you must provide one or more *Purpose Strings*. A purpose string provides isolation between consumers, for example a protector created with a purpose string of "green" would not be able to unprotect data provided by a protector with a purpose of "purple".

: Instances of IDataProtectionProvider and IDataProtector are thread-safe for multiple callers. It is intended that once a component gets a reference to an IDataProtector via a call to CreateProtector, it will use that reference for multiple calls to Protect and Unprotect.

A call to Unprotect will throw CryptographicException if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch CryptographicException instead of swallowing all exceptions.

Consumer APIs

Consumer APIs Overview

The IDataProtectionProvider and IDataProtector interfaces are the basic interfaces through which consumers use the data protection system. They are located in the Microsoft.AspNetCore.DataProtection.Interfaces package.

IDataProtectionProvider   The provider interface represents the root of the data protection system. It cannot directly be used to protect or unprotect data. Instead, the consumer must get a reference to an IDataProtector by calling IDataProtectionProvider.CreateProtector(purpose), where purpose is a string that describes the intended consumer use case. See *Purpose Strings* for much more information on the intent of this parameter and how to choose an appropriate value.

IDataProtector   The protector interface is returned by a call to CreateProtector, and it is this interface which consumers can use to perform protect and unprotect operations.

To protect a piece of data, pass the data to the Protect method. The basic interface defines a method which converts byte[] -> byte[], but there is also an overload (provided as an extension method) which converts string -> string. The security offered by the two methods is identical; the developer should choose whichever overload is most convenient for his use case. Irrespective of the overload chosen, the value returned by the Protect method is now protected (enciphered and tamper-proofed), and the application can send it to an untrusted client.

To unprotect a previously-protected piece of data, pass the protected data to the Unprotect method. (There are byte[]-based and string-based overloads for developer convenience.) If the protected payload was generated by an earlier call to Protect on this same IDataProtector, the Unprotect method will return the original unprotected payload. If the protected payload has been tampered with or was produced by a different IDataProtector, the Unprotect method will throw CryptographicException.

The concept of same vs. different IDataProtector ties back to the concept of purpose. If two IDataProtector instances were generated from the same root IDataProtectionProvider but via different purpose strings in the call to IDataProtectionProvider.CreateProtector, then they are considered *different protectors*, and one will not be able to unprotect payloads generated by the other.

Consuming these interfaces   For a DI-aware component, the intended usage is that the component take an IDataProtectionProvider parameter in its constructor and that the DI system automatically provides this service when the component is instantiated.

:   Some applications (such as console applications or ASP.NET 4.x applications) might not be DI-aware so cannot use the mechanism described here. For these scenarios consult the *Non DI Aware Scenarios* document for more information on getting an instance of an IDataProtection provider without going through DI.

The following sample demonstrates three concepts:

1. *Adding the data protection system* to the service container,

2. Using DI to receive an instance of an IDataProtectionProvider, and

3. Creating an IDataProtector from an IDataProtectionProvider and using it to protect and unprotect data.

```csharp
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

```

```
14          // create an instance of MyClass using the service provider
15          var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
16          instance.RunSample();
17      }
18
19      public class MyClass
20      {
21          IDataProtector _protector;
22
23          // the 'provider' parameter is provided by DI
24          public MyClass(IDataProtectionProvider provider)
25          {
26              _protector = provider.CreateProtector("Contoso.MyClass.v1");
27          }
28
29          public void RunSample()
30          {
31              Console.Write("Enter input: ");
32              string input = Console.ReadLine();
33
34              // protect the payload
35              string protectedPayload = _protector.Protect(input);
36              Console.WriteLine($"Protect returned: {protectedPayload}");
37
38              // unprotect the payload
39              string unprotectedPayload = _protector.Unprotect(protectedPayload);
40              Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
41          }
42      }
43  }
44
45  /*
46   * SAMPLE OUTPUT
47   *
48   * Enter input: Hello world!
49   * Protect returned: CfDJ8ICcgQwZZhlAlTZT...OdfH66i1PnGmpCR5e441xQ
50   * Unprotect returned: Hello world!
51   */
```

The package Microsoft.AspNetCore.DataProtection.Abstractions contains an extension method IService-Provider.GetDataProtector as a developer convenience. It encapsulates as a single operation both retrieving an IDataProtectionProvider from the service provider and calling IDataProtectionProvider.CreateProtector. The following sample demonstrates its usage.

```
1  using System;
2  using Microsoft.AspNetCore.DataProtection;
3  using Microsoft.Extensions.DependencyInjection;
4
5  public class Program
6  {
7      public static void Main(string[] args)
8      {
9          // add data protection services
10         var serviceCollection = new ServiceCollection();
11         serviceCollection.AddDataProtection();
12         var services = serviceCollection.BuildServiceProvider();
13
14         // get an IDataProtector from the IServiceProvider
```

```
15          var protector = services.GetDataProtector("Contoso.Example.v2");
16          Console.Write("Enter input: ");
17          string input = Console.ReadLine();
18
19          // protect the payload
20          string protectedPayload = protector.Protect(input);
21          Console.WriteLine($"Protect returned: {protectedPayload}");
22
23          // unprotect the payload
24          string unprotectedPayload = protector.Unprotect(protectedPayload);
25          Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
26      }
27  }
```

: Instances of IDataProtectionProvider and IDataProtector are thread-safe for multiple callers. It is intended that once a component gets a reference to an IDataProtector via a call to CreateProtector, it will use that reference for multiple calls to Protect and Unprotect.

A call to Unprotect will throw CryptographicException if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch CryptographicException instead of swallowing all exceptions.

Purpose Strings

Components which consume IDataProtectionProvider must pass a unique *purposes* parameter to the CreateProtector method. The purposes *parameter* is inherent to the security of the data protection system, as it provides isolation between cryptographic consumers, even if the root cryptographic keys are the same.

When a consumer specifies a purpose, the purpose string is used along with the root cryptographic keys to derive cryptographic subkeys unique to that consumer. This isolates the consumer from all other cryptographic consumers in the application: no other component can read its payloads, and it cannot read any other component's payloads. This isolation also renders infeasible entire categories of attack against the component.

In the diagram above IDataProtector instances A and B **cannot** read each other's payloads, only their own.

The purpose string doesn't have to be secret. It should simply be unique in the sense that no other well-behaved component will ever provide the same purpose string.

: Using the namespace and type name of the component consuming the data protection APIs is a good rule of thumb, as in practice this information will never conflict.

A Contoso-authored component which is responsible for minting bearer tokens might use Contoso.Security.BearerToken as its purpose string. Or - even better - it might use Contoso.Security.BearerToken.v1 as its purpose string. Appending the version number allows a future version to use Contoso.Security.BearerToken.v2 as its purpose, and the different versions would be completely isolated from one another as far as payloads go.

Since the purposes parameter to CreateProtector is a string array, the above could have been instead specified as ["Contoso.Security.BearerToken", "v1"]. This allows establishing a hierarchy of purposes and opens up the possibility of multi-tenancy scenarios with the data protection system.

> : Components should not allow untrusted user input to be the sole source of input for the purposes chain.
> For example, consider a component Contoso.Messaging.SecureMessage which is responsible for storing secure messages. If the secure messaging component were to call CreateProtector([username]), then a malicious user might create an account with username "Contoso.Security.BearerToken" in an attempt to get the component to call CreateProtector(["Contoso.Security.BearerToken"]), thus inadvertently causing the secure messaging system to mint payloads that could be perceived as authentication tokens.
> A better purposes chain for the messaging component would be CreateProtector(["Contoso.Messaging.SecureMessage", "User: username"]), which provides proper isolation.

The isolation provided by and behaviors of IDataProtectionProvider, IDataProtector, and purposes are as follows:

- For a given IDataProtectionProvider object, the CreateProtector method will create an IDataProtector object uniquely tied to both the IDataProtectionProvider object which created it and the purposes parameter which was passed into the method.

- The purpose parameter must not be null. (If purposes is specified as an array, this means that the array must not be of zero length and all elements of the array must be non-null.) An empty string purpose is technically allowed but is discouraged.

- Two purposes arguments are equivalent if and only if they contain the same strings (using an ordinal comparer) in the same order. A single purpose argument is equivalent to the corresponding single-element purposes array.

- Two IDataProtector objects are equivalent if and only if they are created from equivalent IDataProtectionProvider objects with equivalent purposes parameters.

- For a given IDataProtector object, a call to Unprotect(protectedData) will return the original unprotectedData if and only if protectedData := Protect(unprotectedData) for an equivalent IDataProtector object.

: We're not considering the case where some component intentionally chooses a purpose string which is known to conflict with another component. Such a component would essentially be considered malicious, and this system is not intended to provide security guarantees in the event that malicious code is already running inside of the worker process.

Purpose hierarchy and multi-tenancy

Since an IDataProtector is also implicitly an IDataProtectionProvider, purposes can be chained together. In this sense provider.CreateProtector(["purpose1", "purpose2"]) is equivalent to provider.CreateProtector("purpose1").CreateProtector("purpose2").

This allows for some interesting hierarchical relationships through the data protection system. In the earlier example of *Contoso.Messaging.SecureMessage*, the SecureMessage component can call provider.CreateProtector("Contoso.Messaging.SecureMessage") once upfront and cache the result into a private

_myProvider field. Future protectors can then be created via calls to _myProvider.CreateProtector("User: username"), and these protectors would be used for securing the individual messages.

This can also be flipped. Consider a single logical application which hosts multiple tenants (a CMS seems reasonable), and each tenant can be configured with its own authentication and state management system. The umbrella application has a single master provider, and it calls provider.CreateProtector("Tenant 1") and provider.CreateProtector("Tenant 2") to give each tenant its own isolated slice of the data protection system. The tenants could then derive their own individual protectors based on their own needs, but no matter how hard they try they cannot create protectors which collide with any other tenant in the system. Graphically this is represented as below.
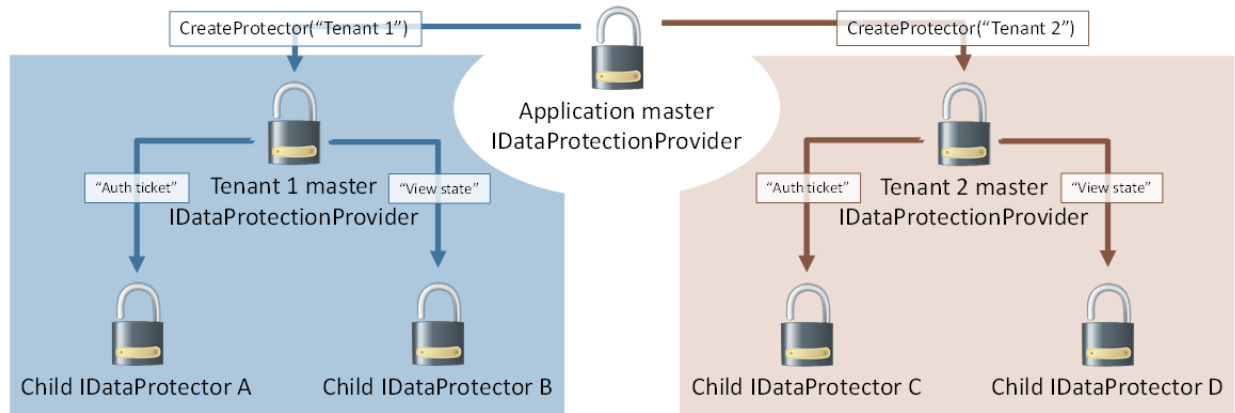
> **:** This assumes the umbrella application controls which APIs are available to individual tenants and that tenants cannot execute arbitrary code on the server. If a tenant can execute arbitrary code, he could perform private reflection to break the isolation guarantees, or he could just read the master keying material directly and derive whatever subkeys he desires.

The data protection system actually uses a sort of multi-tenancy in its default out-of-the-box configuration. By default master keying material is stored in the worker process account's user profile folder (or the registry, for IIS application pool identities). But it is actually fairly common to use a single account to run multiple applications, and thus all these applications would end up sharing the master keying material. To solve this, the data protection system automatically inserts a unique-per-application identifier as the first element in the overall purpose chain. This implicit purpose serves to *isolate individual applications* from one another by effectively treating each application as a unique tenant within the system, and the protector creation process looks identical to the image above.

Password Hashing

The data protection code base includes a package *Microsoft.AspNetCore.Cryptography.KeyDerivation* which contains cryptographic key derivation functions. This package is a standalone component and has no dependencies on the rest of the data protection system. It can be used completely independently. The source exists alongside the data protection code base as a convenience.

The package currently offers a method `KeyDerivation.Pbkdf2` which allows hashing a password using the PBKDF2 algorithm. This API is very similar to the .NET Framework's existing Rfc2898DeriveBytes type, but there are three important distinctions:

1. The `KeyDerivation.Pbkdf2` method supports consuming multiple PRFs (currently `HMACSHA1`, `HMACSHA256`, and `HMACSHA512`), whereas the `Rfc2898DeriveBytes` type only supports `HMACSHA1`.

2. The `KeyDerivation.Pbkdf2` method detects the current operating system and attempts to choose the most optimized implementation of the routine, providing much better performance in certain cases. (On Windows 8, it offers around 10x the throughput of `Rfc2898DeriveBytes`.)

3. The `KeyDerivation.Pbkdf2` method requires the caller to specify all parameters (salt, PRF, and iteration count). The `Rfc2898DeriveBytes` type provides default values for these.

```
using System;
using System.Security.Cryptography;
using Microsoft.AspNetCore.Cryptography.KeyDerivation;

public class Program
{
    public static void Main(string[] args)
    {
        Console.Write("Enter a password: ");
        string password = Console.ReadLine();

        // generate a 128-bit salt using a secure PRNG
        byte[] salt = new byte[128 / 8];
        using (var rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(salt);
        }
        Console.WriteLine($"Salt: {Convert.ToBase64String(salt)}");

        // derive a 256-bit subkey (use HMACSHA1 with 10,000 iterations)
        string hashed = Convert.ToBase64String(KeyDerivation.Pbkdf2(
            password: password,
            salt: salt,
            prf: KeyDerivationPrf.HMACSHA1,
            iterationCount: 10000,
            numBytesRequested: 256 / 8));
        Console.WriteLine($"Hashed: {hashed}");
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter a password: Xtw9NMgx
 * Salt: NZsP6NnmfBuYeJrrAKNuVQ==
 * Hashed: /OOoOer10+tGwTRDTrQSoeCxVTFr6dtYly7d0cPxIak=
 */
```

See the source code for ASP.NET Core Identity's `PasswordHasher` type for a real-world use case.

Limiting the lifetime of protected payloads

There are scenarios where the application developer wants to create a protected payload that expires after a set period of time. For instance, the protected payload might represent a password reset token that should only be valid for one hour. It is certainly possible for the developer to create his own payload format that contains an embedded expiration date, and advanced developers may wish to do this anyway, but for the majority of developers managing these expirations can grow tedious.

To make this easier for our developer audience, the package Microsoft.AspNetCore.DataProtection.Extensions contains utility APIs for creating payloads that automatically expire after a set period of time. These APIs hang off of the ITimeLimitedDataProtector type.

API usage   The ITimeLimitedDataProtector interface is the core interface for protecting and unprotecting time-limited / self-expiring payloads. To create an instance of an ITimeLimitedDataProtector, you'll first need an instance of a regular *IDataProtector* constructed with a specific purpose. Once the IDataProtector instance is available, call the IDataProtector.ToTimeLimitedDataProtector extension method to get back a protector with built-in expiration capabilities.

ITimeLimitedDataProtector exposes the following API surface and extension methods:

- CreateProtector(string purpose) : ITimeLimitedDataProtector This API is similar to the existing IDataProtectionProvider.CreateProtector in that it can be used to create *purpose chains* from a root time-limited protector.

- Protect(byte[] plaintext, DateTimeOffset expiration) : byte[]

- Protect(byte[] plaintext, TimeSpan lifetime) : byte[]

- Protect(byte[] plaintext) : byte[]

- Protect(string plaintext, DateTimeOffset expiration) : string

- Protect(string plaintext, TimeSpan lifetime) : string

- Protect(string plaintext) : string

In addition to the core Protect methods which take only the plaintext, there are new overloads which allow specifying the payload's expiration date. The expiration date can be specified as an absolute date (via a DateTimeOffset) or as a relative time (from the current system time, via a TimeSpan). If an overload which doesn't take an expiration is called, the payload is assumed never to expire.

- Unprotect(byte[] protectedData, out DateTimeOffset expiration) : byte[]

- Unprotect(byte[] protectedData) : byte[]

- Unprotect(string protectedData, out DateTimeOffset expiration) : string

- Unprotect(string protectedData) : string

The Unprotect methods return the original unprotected data. If the payload hasn't yet expired, the absolute expiration is returned as an optional out parameter along with the original unprotected data. If the payload is expired, all overloads of the Unprotect method will throw CryptographicException.

> **:**   It is not advised to use these APIs to protect payloads which require long-term or indefinite persistence. "Can I afford for the protected payloads to be permanently unrecoverable after a month?" can serve as a good rule of thumb; if the answer is no then developers should consider alternative APIs.

The sample below uses the *non-DI code paths* for instantiating the data protection system. To run this sample, ensure that you have first added a reference to the Microsoft.AspNetCore.DataProtection.Extensions package.

```
1   using System;
2   using System.IO;
3   using System.Threading;
4   using Microsoft.AspNetCore.DataProtection;
5
6   public class Program
7   {
8       public static void Main(string[] args)
9       {
10          // create a protector for my application
11
12          var provider = DataProtectionProvider.Create(new DirectoryInfo(@"c:\myapp-keys\"));
13          var baseProtector = provider.CreateProtector("Contoso.TimeLimitedSample");
14
```

```
15          // convert the normal protector into a time-limited protector
16          var timeLimitedProtector = baseProtector.ToTimeLimitedDataProtector();
17
18          // get some input and protect it for five seconds
19          Console.Write("Enter input: ");
20          string input = Console.ReadLine();
21          string protectedData = timeLimitedProtector.Protect(input, lifetime: TimeSpan.FromSeconds(5));
22          Console.WriteLine($"Protected data: {protectedData}");
23
24          // unprotect it to demonstrate that round-tripping works properly
25          string roundtripped = timeLimitedProtector.Unprotect(protectedData);
26          Console.WriteLine($"Round-tripped data: {roundtripped}");
27
28          // wait 6 seconds and perform another unprotect, demonstrating that the payload self-expires
29          Console.WriteLine("Waiting 6 seconds...");
30          Thread.Sleep(6000);
31          timeLimitedProtector.Unprotect(protectedData);
32      }
33  }
34
35  /*
36   * SAMPLE OUTPUT
37   *
38   * Enter input: Hello!
39   * Protected data: CfDJ8Hu5z0zwxn...nLk7Ok
40   * Round-tripped data: Hello!
41   * Waiting 6 seconds...
42   * <<throws CryptographicException with message 'The payload expired at ...'>>
43
44   */
```

Unprotecting payloads whose keys have been revoked

The ASP.NET Core data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like Windows CNG DPAPI and Azure Rights Management are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there is nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data. Keys are never removed from the key ring, so IDataProtector.Unprotect can always recover existing payloads as long as the keys are available and valid.

However, an issue arises when the developer tries to unprotect data that has been protected with a revoked key, as IDataProtector.Unprotect will throw an exception in this case. This might be fine for short-lived or transient payloads (like authentication tokens), as these kinds of payloads can easily be recreated by the system, and at worst the site visitor might be required to log in again. But for persisted payloads, having Unprotect throw could lead to unacceptable data loss.

IPersistedDataProtector To support the scenario of allowing payloads to be unprotected even in the face of revoked keys, the data protection system contains an IPersistedDataProtector type. To get an instance of IPersistedDataProtector, simply get an instance of IDataProtector in the normal fashion and try casting the IDataProtector to IPersistedDataProtector.

: Not all IDataProtector instances can be cast to IPersistedDataProtector. Developers should use the C# as operator or similar to avoid runtime exceptions caused by invalid casts, and they should be prepared to handle the failure case

appropriately.

IPersistedDataProtector exposes the following API surface:

```
DangerousUnprotect(byte[] protectedData, bool ignoreRevocationErrors,
  out bool requiresMigration, out bool wasRevoked) : byte[]
```

This API takes the protected payload (as a byte array) and returns the unprotected payload. There is no string-based overload. The two out parameters are as follows.

- requiresMigration: will be set to true if the key used to protect this payload is no longer the active default key, e.g., the key used to protect this payload is old and a key rolling operation has since taken place. The caller may wish to consider reprotecting the payload depending on his business needs.

- wasRevoked: will be set to true if the key used to protect this payload was revoked.

> : Exercise extreme caution when passing ignoreRevocationErrors: true to the DangerousUnprotect method. If after calling this method the wasRevoked value is true, then the key used to protect this payload was revoked, and the payload's authenticity should be treated as suspect. In this case only continue operating on the unprotected payload if you have some separate assurance that it is authentic, e.g. that it's coming from a secure database rather than being sent by an untrusted web client.

```
1   using System;
2   using System.IO;
3   using System.Text;
4   using Microsoft.AspNetCore.DataProtection;
5   using Microsoft.AspNetCore.DataProtection.KeyManagement;
6   using Microsoft.Extensions.DependencyInjection;
7
8   public class Program
9   {
10      public static void Main(string[] args)
11      {
12          var serviceCollection = new ServiceCollection();
13          serviceCollection.AddDataProtection()
14              // point at a specific folder and use DPAPI to encrypt keys
15              .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
16              .ProtectKeysWithDpapi();
17          var services = serviceCollection.BuildServiceProvider();
18
19          // get a protector and perform a protect operation
20          var protector = services.GetDataProtector("Sample.DangerousUnprotect");
21          Console.Write("Input: ");
22          byte[] input = Encoding.UTF8.GetBytes(Console.ReadLine());
23          var protectedData = protector.Protect(input);
24          Console.WriteLine($"Protected payload: {Convert.ToBase64String(protectedData)}");
25
26          // demonstrate that the payload round-trips properly
27          var roundTripped = protector.Unprotect(protectedData);
28          Console.WriteLine($"Round-tripped payload: {Encoding.UTF8.GetString(roundTripped)}");
29
30          // get a reference to the key manager and revoke all keys in the key ring
31          var keyManager = services.GetService<IKeyManager>();
32          Console.WriteLine("Revoking all keys in the key ring...");
33          keyManager.RevokeAllKeys(DateTimeOffset.Now, "Sample revocation.");
34
```

```
35          // try calling Protect - this should throw
36          Console.WriteLine("Calling Unprotect...");
37          try
38          {
39              var unprotectedPayload = protector.Unprotect(protectedData);
40              Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
41          }
42          catch (Exception ex)
43          {
44              Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
45          }
46
47          // try calling DangerousUnprotect
48          Console.WriteLine("Calling DangerousUnprotect...");
49          try
50          {
51              IPersistedDataProtector persistedProtector = protector as IPersistedDataProtector;
52              if (persistedProtector == null)
53              {
54                  throw new Exception("Can't call DangerousUnprotect.");
55              }
56
57              bool requiresMigration, wasRevoked;
58              var unprotectedPayload = persistedProtector.DangerousUnprotect(
59                  protectedData: protectedData,
60                  ignoreRevocationErrors: true,
61                  requiresMigration: out requiresMigration,
62                  wasRevoked: out wasRevoked);
63              Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
64              Console.WriteLine($"Requires migration = {requiresMigration}, was revoked = {wasRevoked}"
65          }
66          catch (Exception ex)
67          {
68              Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
69          }
70      }
71  }
72
73  /*
74   * SAMPLE OUTPUT
75   *
76   * Input: Hello!
77   * Protected payload: CfDJ8LHIzUCX1ZVBn2BZ...
78   * Round-tripped payload: Hello!
79   * Revoking all keys in the key ring...
80   * Calling Unprotect...
81   * CryptographicException: The key {...} has been revoked.
82   * Calling DangerousUnprotect...
83   * Unprotected payload: Hello!
84   * Requires migration = True, was revoked = True
85   */
```

Configuration

Configuring Data Protection

When the data protection system is initialized it applies some *default settings* based on the operational environment. These settings are generally good for applications running on a single machine. There are some cases where a developer may want to change these (perhaps because his application is spread across multiple machines or for compliance reasons), and for these scenarios the data protection system offers a rich configuration API. There is an extension method AddDataProtection which returns an IDataProtectionBuilder which itself exposes extension methods that you can chain together to configure various data protection options. For instance, to store keys at a UNC share instead of %LOCALAPPDATA% (the default), configure the system as follows:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"));

}
```

> **:** If you change the key persistence location, the system will no longer automatically encrypt keys at rest since it doesn't know whether DPAPI is an appropriate encryption mechanism.

You can configure the system to protect keys at rest by calling any of the ProtectKeysWith* configuration APIs. Consider the example below, which stores keys at a UNC share and encrypts those keys at rest with a specific X.509 certificate.

```
public void ConfigureServices(IServiceCollection services)
  {
      services.AddDataProtection()
          .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"))
          .ProtectKeysWithCertificate("thumbprint");
  }
```

See *key encryption at rest* for more examples and for discussion on the built-in key encryption mechanisms.

To configure the system to use a default key lifetime of 14 days instead of 90 days, consider the following example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
}
```

By default the data protection system isolates applications from one another, even if they're sharing the same physical key repository. This prevents the applications from understanding each other's protected payloads. To share protected payloads between two different applications, configure the system passing in the same application name for both applications as in the below example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .SetApplicationName("my application");
}
```

Finally, you may have a scenario where you do not want an application to automatically roll keys as they approach expiration. One example of this might be applications set up in a primary / secondary relationship, where only the primary application is responsible for key management concerns, and all secondary applications simply have a read-only view of the key ring. The secondary applications can be configured to treat the key ring as read-only by configuring the system as below:

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

Per-application isolation   When the data protection system is provided by an ASP.NET Core host, it will automatically isolate applications from one another, even if those applications are running under the same worker process account and are using the same master keying material. This is somewhat similar to the IsolateApps modifier from System.Web's <machineKey> element.

The isolation mechanism works by considering each application on the local machine as a unique tenant, thus the IDataProtector rooted for any given application automatically includes the application ID as a discriminator. The application's unique ID comes from one of two places.

1. If the application is hosted in IIS, the unique identifier is the application's configuration path. If an application is deployed in a farm environment, this value should be stable assuming that the IIS environments are configured similarly across all machines in the farm.

2. If the application is not hosted in IIS, the unique identifier is the physical path of the application.

The unique identifier is designed to survive resets - both of the individual application and of the machine itself.

This isolation mechanism assumes that the applications are not malicious. A malicious application can always impact any other application running under the same worker process account. In a shared hosting environment where applications are mutually untrusted, the hosting provider should take steps to ensure OS-level isolation between applications, including separating the applications' underlying key repositories.

If the data protection system is not provided by an ASP.NET Core host (e.g., if the developer instantiates it himself via the DataProtectionProvider concrete type), application isolation is disabled by default, and all applications backed by the same keying material can share payloads as long as they provide the appropriate purposes. To provide application isolation in this environment, call the SetApplicationName method on the configuration object, see the *code sample* above.

Changing algorithms   The data protection stack allows changing the default algorithm used by newly-generated keys. The simplest way to do this is to call UseCryptographicAlgorithms from the configuration callback, as in the below example.

```
services.AddDataProtection()
    .UseCryptographicAlgorithms(new AuthenticatedEncryptionSettings()
    {
        EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
        ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
    });
```

The default EncryptionAlgorithm and ValidationAlgorithm are AES-256-CBC and HMACSHA256, respectively. The default policy can be set by a system administrator via *Machine Wide Policy*, but an explicit call to UseCryptographicAlgorithms will override the default policy.

Calling UseCryptographicAlgorithms will allow the developer to specify the desired algorithm (from a predefined built-in list), and the developer does not need to worry about the implementation of the algorithm. For instance, in the

scenario above the data protection system will attempt to use the CNG implementation of AES if running on Windows, otherwise it will fall back to the managed System.Security.Cryptography.Aes class.

The developer can manually specify an implementation if desired via a call to UseCustomCryptographicAlgorithms, as show in the below examples.

: Changing algorithms does not affect existing keys in the key ring. It only affects newly-generated keys.

Specifying custom managed algorithms   To specify custom managed algorithms, create a ManagedAuthenticatedEncryptionSettings instance that points to the implementation types.

```
serviceCollection.AddDataProtection()
    .UseCustomCryptographicAlgorithms(new ManagedAuthenticatedEncryptionSettings()
    {
        // a type that subclasses SymmetricAlgorithm
        EncryptionAlgorithmType = typeof(Aes),

        // specified in bits
        EncryptionAlgorithmKeySize = 256,

        // a type that subclasses KeyedHashAlgorithm
        ValidationAlgorithmType = typeof(HMACSHA256)
    });
```

Generally the *Type properties must point to concrete, instantiable (via a public parameterless ctor) implementations of SymmetricAlgorithm and KeyedHashAlgorithm, though the system special-cases some values like typeof(Aes) for convenience.

:   The SymmetricAlgorithm must have a key length of 128 bits and a block size of 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The KeyedHashAlgorithm must have a digest size of >= 128 bits, and it must support keys of length equal to the hash algorithm's digest length. The KeyedHashAlgorithm is not strictly required to be HMAC.

Specifying custom Windows CNG algorithms   To specify a custom Windows CNG algorithm using CBC-mode encryption + HMAC validation, create a CngCbcAuthenticatedEncryptionSettings instance that contains the algorithmic information.

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(new CngCbcAuthenticatedEncryptionSettings()
    {
        // passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // specified in bits
        EncryptionAlgorithmKeySize = 256,

        // passed to BCryptOpenAlgorithmProvider
        HashAlgorithm = "SHA256",
        HashAlgorithmProvider = null
    });
```

: The symmetric block cipher algorithm must have a key length of 128 bits and a block size of 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The hash algorithm must have a digest size of >= 128 bits and must support being opened with the BCRYPT_ALG_HANDLE_HMAC_FLAG flag. The *Provider properties can be set to null to use the default provider for the specified algorithm. See the BCryptOpenAlgorithmProvider documentation for more information.

To specify a custom Windows CNG algorithm using Galois/Counter Mode encryption + validation, create a CngGcmAuthenticatedEncryptionSettings instance that contains the algorithmic information.

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(new CngGcmAuthenticatedEncryptionSettings()
    {
        // passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // specified in bits
        EncryptionAlgorithmKeySize = 256
    });
});
```

: The symmetric block cipher algorithm must have a key length of 128 bits and a block size of exactly 128 bits, and it must support GCM encryption. The EncryptionAlgorithmProvider property can be set to null to use the default provider for the specified algorithm. See the BCryptOpenAlgorithmProvider documentation for more information.

Specifying other custom algorithms Though not exposed as a first-class API, the data protection system is extensible enough to allow specifying almost any kind of algorithm. For example, it is possible to keep all keys contained within an HSM and to provide a custom implementation of the core encryption and decryption routines. See IAuthenticatedEncryptorConfiguration in the core cryptography extensibility section for more information.

See also *Non DI Aware Scenarios*

Machine Wide Policy

Default Settings

Key Management The system tries to detect its operational environment and provide good zero-configuration behavioral defaults. The heuristic used is as follows.

1. If the system is being hosted in Azure Web Sites, keys are persisted to the "%HOME%\ASP.NET\DataProtection-Keys" folder. This folder is backed by network storage and is synchronized across all machines hosting the application. Keys are not protected at rest.

2. If the user profile is available, keys are persisted to the "%LOCALAPPDATA%\ASP.NET\DataProtection-Keys" folder. Additionally, if the operating system is Windows, they'll be encrypted at rest using DPAPI.

3. If the application is hosted in IIS, keys are persisted to the HKLM registry in a special registry key that is ACLed only to the worker process account. Keys are encrypted at rest using DPAPI.

4. If none of these conditions matches, keys are not persisted outside of the current process. When the process shuts down, all generated keys will be lost.

The developer is always in full control and can override how and where keys are stored. The first three options above should good defaults for most applications similar to how the ASP.NET <machineKey> auto-generation routines worked in the past. The final, fall back option is the only scenario that truly requires the developer to specify *configuration* upfront if he wants key persistence, but this fall-back would only occur in rare situations.

> : If the developer overrides this heuristic and points the data protection system at a specific key repository, automatic encryption of keys at rest will be disabled. At rest protection can be re-enabled via *configuration*.

Key Lifetime   Keys by default have a 90-day lifetime. When a key expires, the system will automatically generate a new key and set the new key as the active key. As long as retired keys remain on the system you will still be able to decrypt any data protected with them. See *key lifetime* for more information.

Default Algorithms   The default payload protection algorithm used is AES-256-CBC for confidentiality and HMAC-SHA256 for authenticity. A 512-bit master key, rolled every 90 days, is used to derive the two sub-keys used for these algorithms on a per-payload basis. See *subkey derivation* for more information.

Machine Wide Policy

When running on Windows, the data protection system has limited support for setting default machine-wide policy for all applications which consume data protection. The general idea is that an administrator might wish to change some default setting (such as algorithms used or key lifetime) without needing to manually update every application on the machine.

> : The system administrator can set default policy, but he cannot enforce it. The application developer can always override any value with one of his own choosing. The default policy only affects applications where the developer has not specified an explicit value for some particular setting.

Setting default policy   To set default policy, an administrator can set known values in the system registry under the following key.

Reg key: `HKLM\SOFTWARE\Microsoft\DotNetPackages\Microsoft.AspNetCore.DataProtection`

If you're on a 64-bit operating system and want to affect the behavior of 32-bit applications, remember also to configure the Wow6432Node equivalent of the above key.

The supported values are:

- EncryptionType [string] - specifies which algorithms should be used for data protection. This value must be "CNG-CBC", "CNG-GCM", or "Managed" and is described in more detail *below*.

- DefaultKeyLifetime [DWORD] - specifies the lifetime for newly-generated keys. This value is specified in days and must be 7.

- KeyEscrowSinks [string] - specifies the types which will be used for key escrow. This value is a semicolon-delimited list of key escrow sinks, where each element in the list is the assembly qualified name of a type which implements IKeyEscrowSink.

Encryption types   If EncryptionType is "CNG-CBC", the system will be configured to use a CBC-mode symmetric block cipher for confidentiality and HMAC for authenticity with services provided by Windows CNG (see *Specifying custom Windows CNG algorithms* for more details). The following additional values are supported, each of which corresponds to a property on the CngCbcAuthenticatedEncryptionSettings type:

- EncryptionAlgorithm [string] - the name of a symmetric block cipher algorithm understood by CNG. This algorithm will be opened in CBC mode.

- EncryptionAlgorithmProvider [string] - the name of the CNG provider implementation which can produce the algorithm EncryptionAlgorithm.

- EncryptionAlgorithmKeySize [DWORD] - the length (in bits) of the key to derive for the symmetric block cipher algorithm.

- HashAlgorithm [string] - the name of a hash algorithm understood by CNG. This algorithm will be opened in HMAC mode.

- HashAlgorithmProvider [string] - the name of the CNG provider implementation which can produce the algorithm HashAlgorithm.

If EncryptionType is "CNG-GCM", the system will be configured to use a Galois/Counter Mode symmetric block cipher for confidentiality and authenticity with services provided by Windows CNG (see *Specifying custom Windows CNG algorithms* for more details). The following additional values are supported, each of which corresponds to a property on the CngGcmAuthenticatedEncryptionSettings type:

- EncryptionAlgorithm [string] - the name of a symmetric block cipher algorithm understood by CNG. This algorithm will be opened in Galois/Counter Mode.

- EncryptionAlgorithmProvider [string] - the name of the CNG provider implementation which can produce the algorithm EncryptionAlgorithm.

- EncryptionAlgorithmKeySize [DWORD] - the length (in bits) of the key to derive for the symmetric block cipher algorithm.

If EncryptionType is "Managed", the system will be configured to use a managed SymmetricAlgorithm for confidentiality and KeyedHashAlgorithm for authenticity (see *Specifying custom managed algorithms* for more details). The following additional values are supported, each of which corresponds to a property on the ManagedAuthenticatedEncryptionSettings type:

- EncryptionAlgorithmType [string] - the assembly-qualified name of a type which implements SymmetricAlgorithm.

- EncryptionAlgorithmKeySize [DWORD] - the length (in bits) of the key to derive for the symmetric encryption algorithm.

- ValidationAlgorithmType [string] - the assembly-qualified name of a type which implements KeyedHashAlgorithm.

If EncryptionType has any other value (other than null / empty), the data protection system will throw an exception at startup.

> **:** When configuring a default policy setting that involves type names (EncryptionAlgorithmType, ValidationAlgorithmType, KeyEscrowSinks), the types must be available to the application. In practice, this means that for applications running on Desktop CLR, the assemblies which contain these types should be GACed. For ASP.NET Core applications running on .NET Core, the packages which contain these types should be referenced in project.json.

Non DI Aware Scenarios

The data protection system is normally designed *to be added to a service container* and to be provided to dependent components via a DI mechanism. However, there may be some cases where this is not feasible, especially when importing the system into an existing application.

To support these scenarios the package Microsoft.AspNetCore.DataProtection.Extensions provides a concrete type DataProtectionProvider which offers a simple way to use the data protection system without going through DI-specific

code paths. The type itself implements IDataProtectionProvider, and constructing it is as easy as providing a DirectoryInfo where this provider's cryptographic keys should be stored.

For example:

```
1   using System;
2   using System.IO;
3   using Microsoft.AspNetCore.DataProtection;
4
5   public class Program
6   {
7       public static void Main(string[] args)
8       {
9           // get the path to %LOCALAPPDATA%\myapp-keys
10          string destFolder = Path.Combine(
11              Environment.GetEnvironmentVariable("LOCALAPPDATA"),
12              "myapp-keys");
13
14          // instantiate the data protection system at this folder
15          var dataProtectionProvider = DataProtectionProvider.Create(
16              new DirectoryInfo(destFolder));
17
18          var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
19          Console.Write("Enter input: ");
20          string input = Console.ReadLine();
21
22          // protect the payload
23          string protectedPayload = protector.Protect(input);
24          Console.WriteLine($"Protect returned: {protectedPayload}");
25
26          // unprotect the payload
27          string unprotectedPayload = protector.Unprotect(protectedPayload);
28          Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
29      }
30  }
31
32  /*
33   * SAMPLE OUTPUT
34   *
35   * Enter input: Hello world!
36   * Protect returned: CfDJ8FWbAn6...ch3hAPm1NJA
37   * Unprotect returned: Hello world!
38   */
```

> **:** By default the DataProtectionProvider concrete type does not encrypt raw key material before persisting it to the file system. This is to support scenarios where the developer points to a network share, in which case the data protection system cannot automatically deduce an appropriate at-rest key encryption mechanism.
> Additionally, the DataProtectionProvider concrete type does not *isolate applications* by default, so all applications pointed at the same key directory can share payloads as long as their purpose parameters match.

The application developer can address both of these if desired. The DataProtectionProvider constructor accepts an *optional configuration callback* which can be used to tweak the behaviors of the system. The sample below demonstrates restoring isolation via an explicit call to SetApplicationName, and it also demonstrates configuring the system to automatically encrypt persisted keys using Windows DPAPI. If the directory points to a UNC share, you may wish to distribute a shared certificate across all relevant machines and to configure the system to use certificate-based encryption instead via a call to *ProtectKeysWithCertificate*.

```
1  using System;
2  using System.IO;
3  using Microsoft.AspNetCore.DataProtection;
4
5  public class Program
6  {
7      public static void Main(string[] args)
8      {
9          // get the path to %LOCALAPPDATA%\myapp-keys
10         string destFolder = Path.Combine(
11             Environment.GetEnvironmentVariable("LOCALAPPDATA"),
12             "myapp-keys");
13
14         // instantiate the data protection system at this folder
15         var dataProtectionProvider = DataProtectionProvider.Create(
16             new DirectoryInfo(destFolder),
17             configuration =>
18             {
19                 configuration.SetApplicationName("my app name");
20                 configuration.ProtectKeysWithDpapi();
21             });
22
23         var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
24         Console.Write("Enter input: ");
25         string input = Console.ReadLine();
26
27         // protect the payload
28         string protectedPayload = protector.Protect(input);
29         Console.WriteLine($"Protect returned: {protectedPayload}");
30
31         // unprotect the payload
32         string unprotectedPayload = protector.Unprotect(protectedPayload);
33         Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
34     }
35 }
```

: Instances of the DataProtectionProvider concrete type are expensive to create. If an application maintains multiple instances of this type and if they're all pointing at the same key storage directory, application performance may be degraded. The intended usage is that the application developer instantiate this type once then keep reusing this single reference as much as possible. The DataProtectionProvider type and all IDataProtector instances created from it are thread-safe for multiple callers.

Extensibility APIs

Core cryptography extensibility

: Types that implement any of the following interfaces should be thread-safe for multiple callers.

IAuthenticatedEncryptor   The **IAuthenticatedEncryptor** interface is the basic building block of the cryptographic subsystem. There is generally one IAuthenticatedEncryptor per key, and the IAuthenticatedEncryptor instance wraps all cryptographic key material and algorithmic information necessary to perform cryptographic operations.

As its name suggests, the type is responsible for providing authenticated encryption and decryption services. It exposes the following two APIs.

- Decrypt(ArraySegment<byte> ciphertext, ArraySegment<byte> additionalAuthenticatedData) : byte[]

- Encrypt(ArraySegment<byte> plaintext, ArraySegment<byte> additionalAuthenticatedData) : byte[]

The Encrypt method returns a blob that includes the enciphered plaintext and an authentication tag. The authentication tag must encompass the additional authenticated data (AAD), though the AAD itself need not be recoverable from the final payload. The Decrypt method validates the authentication tag and returns the deciphered payload. All failures (except ArgumentNullException and similar) should be homogenized to CryptographicException.

: The IAuthenticatedEncryptor instance itself doesn't actually need to contain the key material. For example, the implementation could delegate to an HSM for all operations.

IAuthenticatedEncryptorDescriptor The **IAuthenticatedEncryptorDescriptor** interface represents a type that knows how to create an *IAuthenticatedEncryptor* instance. Its API is as follows.

- CreateEncryptorInstance() : IAuthenticatedEncryptor

- ExportToXml() : XmlSerializedDescriptorInfo

Like IAuthenticatedEncryptor, an instance of IAuthenticatedEncryptorDescriptor is assumed to wrap one specific key. This means that for any given IAuthenticatedEncryptorDescriptor instance, any authenticated encryptors created by its CreateEncryptorInstance method should be considered equivalent, as in the below code sample.

```
// we have an IAuthenticatedEncryptorDescriptor instance
IAuthenticatedEncryptorDescriptor descriptor = ...;

// get an encryptor instance and perform an authenticated encryption operation
ArraySegment<byte> plaintext = new ArraySegment<byte>(Encoding.UTF8.GetBytes("plaintext"));
ArraySegment<byte> aad = new ArraySegment<byte>(Encoding.UTF8.GetBytes("AAD"));
var encryptor1 = descriptor.CreateEncryptorInstance();
byte[] ciphertext = encryptor1.Encrypt(plaintext, aad);

// get another encryptor instance and perform an authenticated decryption operation
var encryptor2 = descriptor.CreateEncryptorInstance();
byte[] roundTripped = encryptor2.Decrypt(new ArraySegment<byte>(ciphertext), aad);


// the 'roundTripped' and 'plaintext' buffers should be equivalent
```

XML Serialization The primary difference between IAuthenticatedEncryptor and IAuthenticatedEncryptorDescriptor is that the descriptor knows how to create the encryptor and supply it with valid arguments. Consider an IAuthenticatedEncryptor whose implementation relies on SymmetricAlgorithm and KeyedHashAlgorithm. The encryptor's job is to consume these types, but it doesn't necessarily know where these types came from, so it can't really write out a proper description of how to recreate itself if the application restarts. The descriptor acts as a higher level on top of this. Since the descriptor knows how to create the encryptor instance (e.g., it knows how to create the required algorithms), it can serialize that knowledge in XML form so that the encryptor instance can be recreated after an application reset. The descriptor can be serialized via its ExportToXml routine. This routine returns an XmlSerializedDescriptorInfo which contains two properties: the XElement representation of the descriptor and the Type which represents an *IAuthenticatedEncryptorDescriptorDeserializer* which can be used to resurrect this descriptor given the corresponding XElement.

The serialized descriptor may contain sensitive information such as cryptographic key material. The data protection system has built-in support for encrypting information before it's persisted to storage. To take advantage of this, the descriptor should mark the element which contains sensitive information with the attribute name "requiresEncryption" (xmlns "http://schemas.asp.net/2015/03/dataProtection"), value "true".

: There's a helper API for setting this attribute. Call the extension method XElement.MarkAsRequiresEncryption() located in namespace Microsoft.AspNetCore.DataProtection.AuthenticatedEncryption.ConfigurationModel.

There can also be cases where the serialized descriptor doesn't contain sensitive information. Consider again the case of a cryptographic key stored in an HSM. The descriptor cannot write out the key material when serializing itself since the HSM will not expose the material in plaintext form. Instead, the descriptor might write out the key-wrapped version of the key (if the HSM allows export in this fashion) or the HSM's own unique identifier for the key.

IAuthenticatedEncryptorDescriptorDeserializer   The **IAuthenticatedEncryptorDescriptorDeserializer** interface represents a type that knows how to deserialize an IAuthenticatedEncryptorDescriptor instance from an XElement. It exposes a single method:

- ImportFromXml(XElement element) : IAuthenticatedEncryptorDescriptor

The ImportFromXml method takes the XElement that was returned by *IAuthenticatedEncryptorDescriptor.ExportToXml* and creates an equivalent of the original IAuthenticatedEncryptorDescriptor.

Types which implement IAuthenticatedEncryptorDescriptorDeserializer should have one of the following two public constructors:

- .ctor(IServiceProvider)

- .ctor()

: The IServiceProvider passed to the constructor may be null.

IAuthenticatedEncryptorConfiguration   The **IAuthenticatedEncryptorConfiguration** interface represents a type which knows how to create *IAuthenticatedEncryptorDescriptor* instances. It exposes a single API.

- CreateNewDescriptor() : IAuthenticatedEncryptorDescriptor

Think of IAuthenticatedEncryptorConfiguration as the top-level factory. The configuration serves as a template. It wraps algorithmic information (e.g., this configuration produces descriptors with an AES-128-GCM master key), but it is not yet associated with a specific key.

When CreateNewDescriptor is called, fresh key material is created solely for this call, and a new IAuthenticatedEncryptorDescriptor is produced which wraps this key material and the algorithmic information required to consume the material. The key material could be created in software (and held in memory), it could be created and held within an HSM, and so on. The crucial point is that any two calls to CreateNewDescriptor should never create equivalent IAuthenticatedEncryptorDescriptor instances.

The IAuthenticatedEncryptorConfiguration type serves as the entry point for key creation routines such as *automatic key rolling*. To change the implementation for all future keys, register a singleton IAuthenticatedEncryptorConfiguration in the service container.

Key management extensibility

: Read the *key management* section before reading this section, as it explains some of the fundamental concepts behind these APIs.

: Types that implement any of the following interfaces should be thread-safe for multiple callers.

Key   The IKey interface is the basic representation of a key in cryptosystem. The term key is used here in the abstract sense, not in the literal sense of "cryptographic key material". A key has the following properties:

- Activation, creation, and expiration dates
- Revocation status
- Key identifier (a GUID)

Additionally, IKey exposes a CreateEncryptorInstance method which can be used to create an *IAuthenticatedEncryptor* instance tied to this key.

: There is no API to retrieve the raw cryptographic material from an IKey instance.

IKeyManager   The IKeyManager interface represents an object responsible for general key storage, retrieval, and manipulation. It exposes three high-level operations:

- Create a new key and persist it to storage.
- Get all keys from storage.
- Revoke one or more keys and persist the revocation information to storage.

: Writing an IKeyManager is a very advanced task, and the majority of developers should not attempt it. Instead, most developers should take advantage of the facilities offered by the *XmlKeyManager* class.

XmlKeyManager   The XmlKeyManager type is the in-box concrete implementation of IKeyManager. It provides several useful facilities, including key escrow and encryption of keys at rest. Keys in this system are represented as XML elements (specifically, XElement).

XmlKeyManager depends on several other components in the course of fulfilling its tasks:

- IAuthenticatedEncryptorConfiguration, which dictates the algorithms used by new keys.
- IXmlRepository, which controls where keys are persisted in storage.
- IXmlEncryptor [optional], which allows encrypting keys at rest.
- IKeyEscrowSink [optional], which provides key escrow services.

Below are high-level diagrams which indicate how these components are wired together within XmlKeyManager.

In the implementation of CreateNewKey, the IAuthenticatedEncryptorConfiguration component is used to create a unique IAuthenticatedEncryptorDescriptor, which is then serialized as XML. If a key escrow sink is present, the raw (unencrypted) XML is provided to the sink for long-term storage. The unencrypted XML is then run through an IXmlEncryptor (if required) to generate the encrypted XML document. This encrypted document is persisted to long-term storage via the IXmlRepository. (If no IXmlEncryptor is configured, the unencrypted document is persisted in the IXmlRepository.)
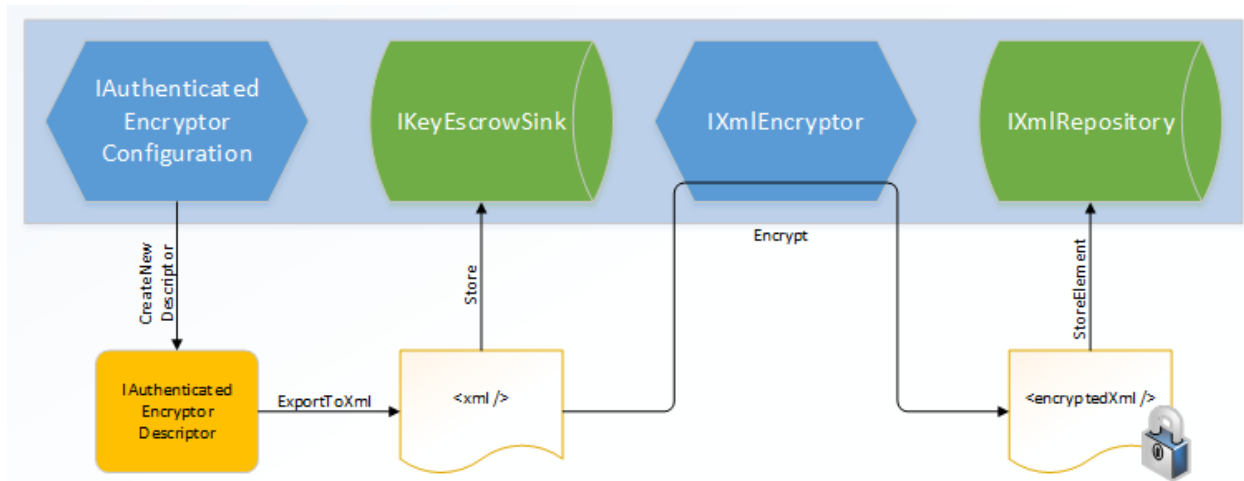
1.1: Key Creation / CreateNewKey
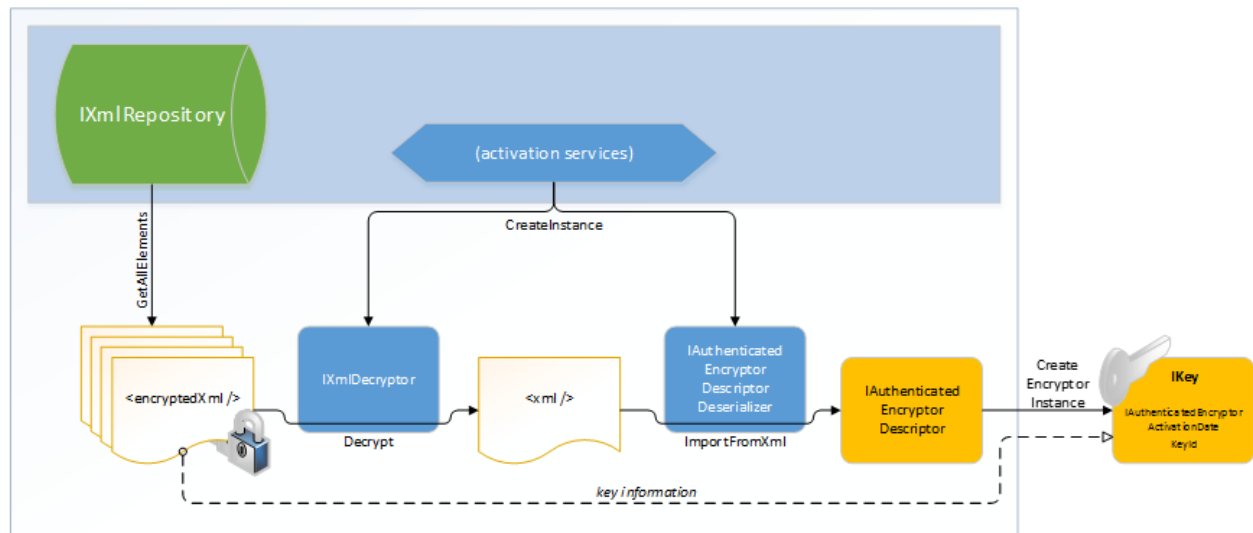
1.2: Key Retrieval / GetAllKeys

In the implementation of GetAllKeys, the XML documents representing keys and revocations are read from the underlying IXmlRepository. If these documents are encrypted, the system will automatically decrypt them. XmlKeyManager creates the appropriate IAuthenticatedEncryptorDescriptorDeserializer instances to deserialize the documents back into IAuthenticatedEncryptorDescriptor instances, which are then wrapped in individual IKey instances. This collection of IKey instances is returned to the caller.

Further information on the particular XML elements can be found in the *key storage format document*.

IXmlRepository   The IXmlRepository interface represents a type that can persist XML to and retrieve XML from a backing store. It exposes two APIs:

- GetAllElements() : IReadOnlyCollection<XElement>

- StoreElement(XElement element, string friendlyName)

Implementations of IXmlRepository don't need to parse the XML passing through them. They should treat the XML documents as opaque and let higher layers worry about generating and parsing the documents.

There are two built-in concrete types which implement IXmlRepository: FileSystemXmlRepository and RegistryXmlRepository. See the *key storage providers document* for more information. Registering a custom IXmlRepository would be the appropriate manner to use a different backing store, e.g., Azure Blob Storage. To change the default repository application-wide, register a custom singleton IXmlRepository in the service provider.

IXmlEncryptor   The IXmlEncryptor interface represents a type that can encrypt a plaintext XML element. It exposes a single API:

- Encrypt(XElement plaintextElement) : EncryptedXmlInfo

If a serialized IAuthenticatedEncryptorDescriptor contains any elements marked as "requires encryption", then XmlKeyManager will run those elements through the configured IXmlEncryptor's Encrypt method, and it will persist the enciphered element rather than the plaintext element to the IXmlRepository. The output of the Encrypt method is an EncryptedXmlInfo object. This object is a wrapper which contains both the resultant enciphered XElement and the Type which represents an IXmlDecryptor which can be used to decipher the corresponding element.

There are four built-in concrete types which implement IXmlEncryptor: CertificateXmlEncryptor, DpapiNGXmlEncryptor, DpapiXmlEncryptor, and NullXmlEncryptor. See the *key encryption at rest document* for more information. To change the default key-encryption-at-rest mechanism application-wide, register a custom singleton IXmlEncryptor in the service provider.

IXmlDecryptor   The IXmlDecryptor interface represents a type that knows how to decrypt an XElement that was enciphered via an IXmlEncryptor. It exposes a single API:

- Decrypt(XElement encryptedElement) : XElement

The Decrypt method undoes the encryption performed by IXmlEncryptor.Encrypt. Generally each concrete IXmlEncryptor implementation will have a corresponding concrete IXmlDecryptor implementation.

Types which implement IXmlDecryptor should have one of the following two public constructors:

- .ctor(IServiceProvider)

- .ctor()

: The IServiceProvider passed to the constructor may be null.

IKeyEscrowSink The IKeyEscrowSink interface represents a type that can perform escrow of sensitive information. Recall that serialized descriptors might contain sensitive information (such as cryptographic material), and this is what led to the introduction of the *IXmlEncryptor* type in the first place. However, accidents happen, and keyrings can be deleted or become corrupted.

The escrow interface provides an emergency escape hatch, allowing access to the raw serialized XML before it is transformed by any configured *IXmlEncryptor*. The interface exposes a single API:

- Store(Guid keyId, XElement element)

It is up to the IKeyEscrowSink implementation to handle the provided element in a secure manner consistent with business policy. One possible implementation could be for the escrow sink to encrypt the XML element using a known corporate X.509 certificate where the certificate's private key has been escrowed; the CertificateXmlEncryptor type can assist with this. The IKeyEscrowSink implementation is also responsible for persisting the provided element appropriately.

By default no escrow mechanism is enabled, though server administrators can *configure this globally*. It can also be configured programmatically via the *IDataProtectionBuilder.AddKeyEscrowSink* method as shown in the sample below. The *AddKeyEscrowSink* method overloads mirror the *IServiceCollection.AddSingleton* and *IServiceCollection.AddInstance* overloads, as IKeyEscrowSink instances are intended to be singletons. If multiple IKeyEscrowSink instances are registered, each one will be called during key generation, so keys can be escrowed to multiple mechanisms simultaneously.

There is no API to read material from an IKeyEscrowSink instance. This is consistent with the design theory of the escrow mechanism: it's intended to make the key material accessible to a trusted authority, and since the application is itself not a trusted authority, it shouldn't have access to its own escrowed material.

The following sample code demonstrates creating and registering an IKeyEscrowSink where keys are escrowed such that only members of "CONTOSODomain Admins" can recover them.

: To run this sample, you must be on a domain-joined Windows 8 / Windows Server 2012 machine, and the domain controller must be Windows Server 2012 or later.

```
using System;
using System.IO;
using System.Xml.Linq;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.AspNetCore.DataProtection.XmlEncryption;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi()
            .AddKeyEscrowSink(sp => new MyKeyEscrowSink(sp));
        var services = serviceCollection.BuildServiceProvider();

        // get a reference to the key manager and force a new key to be generated
        Console.WriteLine("Generating new key...");
        var keyManager = services.GetService<IKeyManager>();
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
```

```
                expirationDate: DateTimeOffset.Now.AddDays(7));
    }

    // A key escrow sink where keys are escrowed such that they
    // can be read by members of the CONTOSO\Domain Admins group.
    private class MyKeyEscrowSink : IKeyEscrowSink
    {
        private readonly IXmlEncryptor _escrowEncryptor;

        public MyKeyEscrowSink(IServiceProvider services)
        {
            // Assuming I'm on a machine that's a member of the CONTOSO
            // domain, I can use the Domain Admins SID to generate an
            // encrypted payload that only they can read. Sample SID from
            // https://technet.microsoft.com/en-us/library/cc778824(v=ws.10).aspx.
            _escrowEncryptor = new DpapiNGXmlEncryptor(
                "SID=S-1-5-21-1004336348-1177238915-682003330-512",
                DpapiNGProtectionDescriptorFlags.None,
                services);
        }

        public void Store(Guid keyId, XElement element)
        {
            // Encrypt the key element to the escrow encryptor.
            var encryptedXmlInfo = _escrowEncryptor.Encrypt(element);

            // A real implementation would save the escrowed key to a
            // write-only file share or some other stable storage, but
            // in this sample we'll just write it out to the console.
            Console.WriteLine($"Escrowing key {keyId}");
            Console.WriteLine(encryptedXmlInfo.EncryptedElement);

            // Note: We cannot read the escrowed key material ourselves.
            // We need to get a member of CONTOSO\Domain Admins to read
            // it for us in the event we need to recover it.
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Generating new key...
 * Escrowing key 38e74534-c1b8-4b43-aea1-79e856a822e5
 * <encryptedKey>
 *   <!-- This key is encrypted with Windows DPAPI-NG. -->
 *   <!-- Rule: SID=S-1-5-21-1004336348-1177238915-682003330-512 -->
 *   <value>MIIIfAYJKoZIhvcNAQcDoIIIbTCCCGkCAQ...T5rA4g==</value>
 * </encryptedKey>
 */
```

Miscellaneous APIs

: Types that implement any of the following interfaces should be thread-safe for multiple callers.

ISecret   The ISecret interface represents a secret value, such as cryptographic key material. It contains the following API surface.

- Length : int

- Dispose() : void

- WriteSecretIntoBuffer(ArraySegment<byte> buffer) : void

The WriteSecretIntoBuffer method populates the supplied buffer with the raw secret value. The reason this API takes the buffer as a parameter rather than returning a byte[] directly is that this gives the caller the opportunity to pin the buffer object, limiting secret exposure to the managed garbage collector.

The Secret type is a concrete implementation of ISecret where the secret value is stored in in-process memory. On Windows platforms, the secret value is encrypted via CryptProtectMemory.

Implementation

Authenticated encryption details.

Calls to IDataProtector.Protect are authenticated encryption operations. The Protect method offers both confidentiality and authenticity, and it is tied to the purpose chain that was used to derive this particular IDataProtector instance from its root IDataProtectionProvider.

IDataProtector.Protect takes a byte[] plaintext parameter and produces a byte[] protected payload, whose format is described below. (There is also an extension method overload which takes a string plaintext parameter and returns a string protected payload. If this API is used the protected payload format will still have the below structure, but it will be base64url-encoded.)

Protected payload format   The protected payload format consists of three primary components:

- A 32-bit magic header that identifies the version of the data protection system.

- A 128-bit key id that identifies the key used to protect this particular payload.

- The remainder of the protected payload is *specific to the encryptor encapsulated by this key*. In the example below the key represents an AES-256-CBC + HMACSHA256 encryptor, and the payload is further subdivided as follows: * A 128-bit key modifier. * A 128-bit initialization vector. * 48 bytes of AES-256-CBC output. * An HMACSHA256 authentication tag.

A sample protected payload is illustrated below.

```
1   09 F0 C9 F0 80 9C 81 0C 19 66 19 40 95 36 53 F8
2   AA FF EE 57 57 2F 40 4C 3F 7F CC 9D CC D9 32 3E
3   84 17 99 16 EC BA 1F 4A A1 18 45 1F 2D 13 7A 28
4   79 6B 86 9C F8 B7 84 F9 26 31 FC B1 86 0A F1 56
5   61 CF 14 58 D3 51 6F CF 36 50 85 82 08 2D 3F 73
6   5F B0 AD 9E 1A B2 AE 13 57 90 C8 F5 7C 95 4E 6A
7   8A AA 06 EF 43 CA 19 62 84 7C 11 B2 C8 71 9D AA
8   52 19 2E 5B 4C 1E 54 F0 55 BE 88 92 12 C1 4B 5E
9   52 C9 74 A0
```

From the payload format above the first 32 bits, or 4 bytes are the magic header identifying the version (09 F0 C9 F0)

The next 128 bits, or 16 bytes is the key identifier (80 9C 81 0C 19 66 19 40 95 36 53 F8 AA FF EE 57)

The remainder contains the payload and is specific to the format used.

> **:** All payloads protected to a given key will begin with the same 20-byte (magic value, key id) header. Administrators can use this fact for diagnostic purposes to approximate when a payload was generated. For example, the payload above corresponds to key {0c819c80-6619-4019-9536-53f8aaffee57}. If after checking the key repository you find that this specific key's activation date was 2015-01-01 and its expiration date was 2015-03-01, then it is reasonable to assume that the payload (if not tampered with) was generated within that window, give or take a small fudge factor on either side.

Subkey Derivation and Authenticated Encryption

Most keys in the key ring will contain some form of entropy and will have algorithmic information stating "CBC-mode encryption + HMAC validation" or "GCM encryption + validation". In these cases, we refer to the embedded entropy as the master keying material (or KM) for this key, and we perform a key derivation function to derive the keys that will be used for the actual cryptographic operations.

: Keys are abstract, and a custom implementation might not behave as below. If the key provides its own implementation of IAuthenticatedEncryptor rather than using one of our built-in factories, the mechanism described in this section no longer applies.

Additional authenticated data and subkey derivation   The IAuthenticatedEncryptor interface serves as the core interface for all authenticated encryption operations. Its Encrypt method takes two buffers: plaintext and additionalAuthenticatedData (AAD). The plaintext contents flow unchanged the call to IDataProtector.Protect, but the AAD is generated by the system and consists of three components:

1. The 32-bit magic header 09 F0 C9 F0 that identifies this version of the data protection system.

2. The 128-bit key id.

3. A variable-length string formed from the purpose chain that created the IDataProtector that is performing this operation.

Because the AAD is unique for the tuple of all three components, we can use it to derive new keys from KM instead of using KM itself in all of our cryptographic operations. For every call to IAuthenticatedEncryptor.Encrypt, the following key derivation process takes place:

$(K_E, K_H) = SP800\_108\_CTR\_HMACSHA512(K_M, AAD, contextHeader \| keyModifier)$

Here, we're calling the NIST SP800-108 KDF in Counter Mode (see NIST SP800-108, Sec. 5.1) with the following parameters:

- Key derivation key (KDK) = K_M

- PRF = HMACSHA512

- label = additionalAuthenticatedData

- context = contextHeader ‖ keyModifier

The context header is of variable length and essentially serves as a thumbprint of the algorithms for which we're deriving K_E and K_H. The key modifier is a 128-bit string randomly generated for each call to Encrypt and serves to ensure with overwhelming probability that KE and KH are unique for this specific authentication encryption operation, even if all other input to the KDF is constant.

For CBC-mode encryption + HMAC validation operations, $| K_E |$ is the length of the symmetric block cipher key, and $| K_H |$ is the digest size of the HMAC routine. For GCM encryption + validation operations, $| K_H | = 0$.

CBC-mode encryption + HMAC validation   Once K_E is generated via the above mechanism, we generate a random initialization vector and run the symmetric block cipher algorithm to encipher the plaintext. The initialization vector and ciphertext are then run through the HMAC routine initialized with the key K_H to produce the MAC. This process and the return value is represented graphically below.
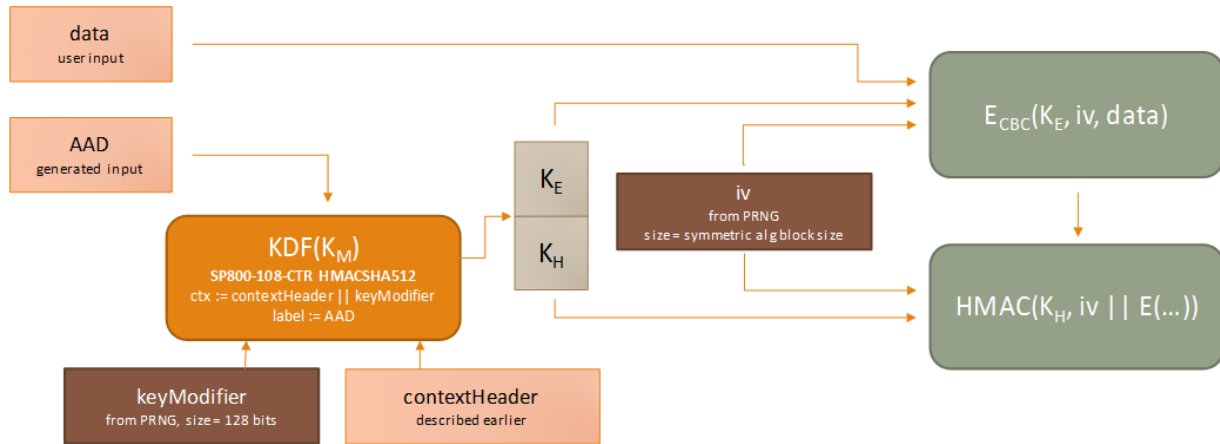
1.3: output:= keyModifier ‖ iv ‖ E_{cbc} (K_E,iv,data) ‖ HMAC(K_H, iv ‖ E_{cbc} (K_E,iv,data))

: The IDataProtector.Protect implementation will *prepend the magic header and key id* to output before returning it to the caller. Because the magic header and key id are implicitly part of *AAD*, and because the key modifier is fed as input to the KDF, this means that every single byte of the final returned payload is authenticated by the MAC.

Galois/Counter Mode encryption + validation   Once K_E is generated via the above mechanism, we generate a random 96-bit nonce and run the symmetric block cipher algorithm to encipher the plaintext and produce the 128-bit authentication tag.
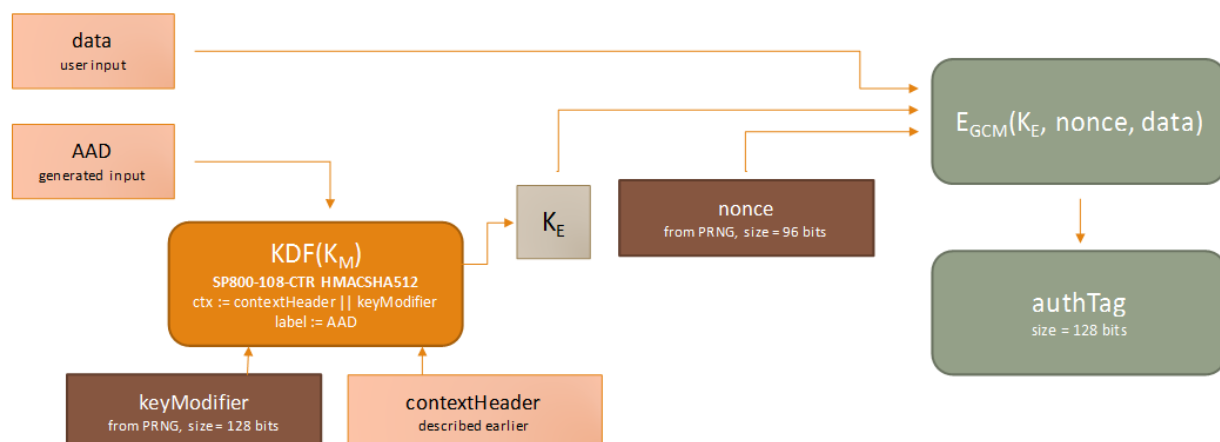
1.4: output := keyModifier ‖ nounce ‖ E_{gcm} (K_E,nounce,data) ‖ authTag

: Even though GCM natively supports the concept of AAD, we're still feeding AAD only to the original KDF, opting

to pass an empty string into GCM for its AAD parameter. The reason for this is two-fold. First, *to support agility* we never want to use K_M directly as the encryption key. Additionally, GCM imposes very strict uniqueness requirements on its inputs. The probability that the GCM encryption routine is ever invoked on two or more distinct sets of input data with the same (key, nonce) pair must not exceed 2^{32}. If we fix K_E we cannot perform more than 2^{32} encryption operations before we run afoul of the 2^{-32} limit. This might seem like a very large number of operations, but a high-traffic web server can go through 4 billion requests in mere days, well within the normal lifetime for these keys. To stay compliant of the 2^{-32} probability limit, we continue to use a 128-bit key modifier and 96-bit nonce, which radically extends the usable operation count for any given K_M. For simplicity of design we share the KDF code path between CBC and GCM operations, and since AAD is already considered in the KDF there is no need to forward it to the GCM routine.

Context headers

Background and theory   In the data protection system, a "key" means an object that can provide authenticated encryption services. Each key is identified by a unique id (a GUID), and it carries with it algorithmic information and entropic material. It is intended that each key carry unique entropy, but the system cannot enforce that, and we also need to account for developers who might change the key ring manually by modifying the algorithmic information of an existing key in the key ring. To achieve our security requirements given these cases the data protection system has a concept of cryptographic agility, which allows securely using a single entropic value across multiple cryptographic algorithms.

Most systems which support cryptographic agility do so by including some identifying information about the algorithm inside the payload. The algorithm's OID is generally a good candidate for this. However, one problem that we ran into is that there are multiple ways to specify the same algorithm: "AES" (CNG) and the managed Aes, AesManaged, AesCryptoServiceProvider, AesCng, and RijndaelManaged (given specific parameters) classes are all actually the same thing, and we'd need to maintain a mapping of all of these to the correct OID. If a developer wanted to provide a custom algorithm (or even another implementation of AES!), he'd have to tell us its OID. This extra registration step makes system configuration particularly painful.

Stepping back, we decided that we were approaching the problem from the wrong direction. An OID tells you what the algorithm is, but we don't actually care about this. If we need to use a single entropic value securely in two different algorithms, it's not necessary for us to know what the algorithms actually are. What we actually care about is how they behave. Any decent symmetric block cipher algorithm is also a strong pseudorandom permutation (PRP): fix the inputs (key, chaining mode, IV, plaintext) and the ciphertext output will with overwhelming probability be distinct from any other symmetric block cipher algorithm given the same inputs. Similarly, any decent keyed hash function is also a strong pseudorandom function (PRF), and given a fixed input set its output will overwhelmingly be distinct from any other keyed hash function.

We use this concept of strong PRPs and PRFs to build up a context header. This context header essentially acts as a stable thumbprint over the algorithms in use for any given operation, and it provides the cryptographic agility needed by the data protection system. This header is reproducible and is used later as part of the *subkey derivation process*. There are two different ways to build the context header depending on the modes of operation of the underlying algorithms.

CBC-mode encryption + HMAC authentication   The context header consists of the following components:

- [16 bits] The value 00 00, which is a marker meaning "CBC encryption + HMAC authentication".
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The key length (in bytes, big-endian) of the HMAC algorithm. (Currently the key size always matches the digest size.)

- [32 bits] The digest size (in bytes, big-endian) of the HMAC algorithm.

- EncCBC(K_E, IV, ""), which is the output of the symmetric block cipher algorithm given an empty string input and where IV is an all-zero vector. The construction of K_E is described below.

- MAC(K_H, ""), which is the output of the HMAC algorithm given an empty string input. The construction of K_H is described below.

Ideally we could pass all-zero vectors for K_E and K_H. However, we want to avoid the situation where the underlying algorithm checks for the existence of weak keys before performing any operations (notably DES and 3DES), which precludes using a simple or repeatable pattern like an all-zero vector.

Instead, we use the NIST SP800-108 KDF in Counter Mode (see NIST SP800-108, Sec. 5.1) with a zero-length key, label, and context and HMACSHA512 as the underlying PRF. We derive | K_E | + | K_H | bytes of output, then decompose the result into K_E and K_H themselves. Mathematically, this is represented as follows.

(K_E || K_H) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")

Example: AES-192-CBC + HMACSHA256 As an example, consider the case where the symmetric block cipher algorithm is AES-192-CBC and the validation algorithm is HMACSHA256. The system would generate the context header using the following steps.

First, let (K_E || K_H) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = ""), where | K_E | = 192 bits and | K_H | = 256 bits per the specified algorithms. This leads to K_E = 5BB6..21DD and K_H = A04A..00A9 in the example below:

```
5B B6 C9 83 13 78 22 1D 8E 10 73 CA CF 65 8E B0
61 62 42 71 CB 83 21 DD A0 4A 05 00 5B AB C0 A2
49 6F A5 61 E3 E2 49 87 AA 63 55 CD 74 0A DA C4
B7 92 3D BF 59 90 00 A9
```

Next, compute Enc$_{CBC}$ (K_E, IV, "") for AES-192-CBC given IV = 0* and K_E as above.

result := F474B1872B3B53E4721DE19C0841DB6F

Next, compute MAC(K_H, "") for HMACSHA256 given K_H as above.

result := D4791184B996092EE1202F36E8608FA8FBD98ABDFF5402F264B1D7211536220C

This produces the full context header below:

```
00 00 00 00 00 18 00 00 00 10 00 00 00 20 00 00
00 20 F4 74 B1 87 2B 3B 53 E4 72 1D E1 9C 08 41
DB 6F D4 79 11 84 B9 96 09 2E E1 20 2F 36 E8 60
8F A8 FB D9 8A BD FF 54 02 F2 64 B1 D7 21 15 36
22 0C
```

This context header is the thumbprint of the authenticated encryption algorithm pair (AES-192-CBC encryption + HMACSHA256 validation). The components, as described *above* are:

- the marker (00 00)

- the block cipher key length (00 00 00 18)

- the block cipher block size (00 00 00 10)

- the HMAC key length (00 00 00 20)

- the HMAC digest size (00 00 00 20)

- the block cipher PRP output (F4 74 - DB 6F) and

- the HMAC PRF output (D4 79 - end).

: The CBC-mode encryption + HMAC authentication context header is built the same way regardless of whether the algorithms implementations are provided by Windows CNG or by managed SymmetricAlgorithm and KeyedHashAlgorithm types. This allows applications running on different operating systems to reliably produce the same context header even though the implementations of the algorithms differ between OSes. (In practice, the KeyedHashAlgorithm doesn't have to be a proper HMAC. It can be any keyed hash algorithm type.)

Example: 3DES-192-CBC + HMACSHA1 First, let (K_E ǁ K_H) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = ""), where | K_E | = 192 bits and | K_H | = 160 bits per the specified algorithms. This leads to K_E = A219..E2BB and K_H = DC4A..B464 in the example below:

```
A2 19 60 2F 83 A9 13 EA B0 61 3A 39 B8 A6 7E 22
61 D9 F8 6C 10 51 E2 BB DC 4A 00 D7 03 A2 48 3E
D1 F7 5A 34 EB 28 3E D7 D4 67 B4 64
```

Next, compute Enc_{CBC} (K_E, IV, "") for 3DES-192-CBC given IV = 0* and K_E as above.

result := ABB100F81E53E10E

Next, compute MAC(K_H, "") for HMACSHA1 given K_H as above.

result := 76EB189B35CF03461DDF877CD9F4B1B4D63A7555

This produces the full context header which is a thumbprint of the authenticated encryption algorithm pair (3DES-192-CBC encryption + HMACSHA1 validation), shown below:

```
00 00 00 00 00 18 00 00 00 08 00 00 00 14 00 00
00 14 AB B1 00 F8 1E 53 E1 0E 76 EB 18 9B 35 CF
03 46 1D DF 87 7C D9 F4 B1 B4 D6 3A 75 55
```

The components break down as follows:

- the marker (00 00)
- the block cipher key length (00 00 00 18)
- the block cipher block size (00 00 00 08)
- the HMAC key length (00 00 00 14)
- the HMAC digest size (00 00 00 14)
- the block cipher PRP output (AB B1 - E1 0E) and
- the HMAC PRF output (76 EB - end).

Galois/Counter Mode encryption + authentication The context header consists of the following components:

- [16 bits] The value 00 01, which is a marker meaning "GCM encryption + authentication".
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The nonce size (in bytes, big-endian) used during authenticated encryption operations. (For our system, this is fixed at nonce size = 96 bits.)
- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm. (For GCM, this is fixed at block size = 128 bits.)
- [32 bits] The authentication tag size (in bytes, big-endian) produced by the authenticated encryption function. (For our system, this is fixed at tag size = 128 bits.)

- [128 bits] The tag of Enc_{GCM} (K_E, nonce, ""), which is the output of the symmetric block cipher algorithm given an empty string input and where nonce is a 96-bit all-zero vector.

K_E is derived using the same mechanism as in the CBC encryption + HMAC authentication scenario. However, since there is no K_H in play here, we essentially have | K_H | = 0, and the algorithm collapses to the below form.

K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")

Example: AES-256-GCM   First, let K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = ""), where | K_E | = 256 bits.

K_E := 22BC6F1B171C08C4AE2F27444AF8FC8B3087A90006CAEA91FDCFB47C1B8733B8

Next, compute the authentication tag of Enc_{GCM} (K_E, nonce, "") for AES-256-GCM given nonce = 0^{96} and K_E as above.

result := E7DCCE66DF855A323A6BB7BD7A59BE45

This produces the full context header below:

```
00 01 00 00 00 20 00 00 00 0C 00 00 00 10 00 00
00 10 E7 DC CE 66 DF 85 5A 32 3A 6B B7 BD 7A 59
BE 45
```

The components break down as follows:

- the marker (00 01)
- the block cipher key length (00 00 00 20)
- the nonce size (00 00 00 0C)
- the block cipher block size (00 00 00 10)
- the authentication tag size (00 00 00 10) and
- the authentication tag from running the block cipher (E7 DC - end).

Key Management

The data protection system automatically manages the lifetime of master keys used to protect and unprotect payloads. Each key can exist in one of four stages.

- Created - the key exists in the key ring but has not yet been activated. The key shouldn't be used for new Protect operations until sufficient time has elapsed that the key has had a chance to propagate to all machines that are consuming this key ring.
- Active - the key exists in the key ring and should be used for all new Protect operations.
- Expired - the key has run its natural lifetime and should no longer be used for new Protect operations.
- Revoked - the key is compromised and must not be used for new Protect operations.

Created, active, and expired keys may all be used to unprotect incoming payloads. Revoked keys by default may not be used to unprotect payloads, but the application developer can *override this behavior* if necessary.

> :   The developer might be tempted to delete a key from the key ring (e.g., by deleting the corresponding file from the file system). At that point, all data protected by the key is permanently undecipherable, and there is no emergency override like there is with revoked keys. Deleting a key is truly destructive behavior, and consequently the data protection system exposes no first-class API for performing this operation.

Default key selection   When the data protection system reads the key ring from the backing repository, it will attempt to locate a "default" key from the key ring. The default key is used for new Protect operations.

The general heuristic is that the data protection system chooses the key with the most recent activation date as the default key. (There's a small fudge factor to allow for server-to-server clock skew.) If the key is expired or revoked, and if the application has not disabled automatic key generation, then a new key will be generated with immediate activation per the *key expiration and rolling* policy below.

The reason the data protection system generates a new key immediately rather than falling back to a different key is that new key generation should be treated as an implicit expiration of all keys that were activated prior to the new key. The general idea is that new keys may have been configured with different algorithms or encryption-at-rest mechanisms than old keys, and the system should prefer the current configuration over falling back.

There is an exception. If the application developer has *disabled automatic key generation*, then the data protection system must choose something as the default key. In this fallback scenario, the system will choose the non-revoked key with the most recent activation date, with preference given to keys that have had time to propagate to other machines in the cluster. The fallback system may end up choosing an expired default key as a result. The fallback system will never choose a revoked key as the default key, and if the key ring is empty or every key has been revoked then the system will produce an error upon initialization.

Key expiration and rolling   When a key is created, it is automatically given an activation date of { now + 2 days } and an expiration date of { now + 90 days }. The 2-day delay before activation gives the key time to propagate through the system. That is, it allows other applications pointing at the backing store to observe the key at their next auto-refresh period, thus maximizing the chances that when the key ring does become active it has propagated to all applications that might need to use it.

If the default key will expire within 2 days and if the key ring does not already have a key that will be active upon expiration of the default key, then the data protection system will automatically persist a new key to the key ring. This new key has an activation date of { default key's expiration date } and an expiration date of { now + 90 days }. This allows the system to automatically roll keys on a regular basis with no interruption of service.

There might be circumstances where a key will be created with immediate activation. One example would be when the application hasn't run for a time and all keys in the key ring are expired. When this happens, the key is given an activation date of { now } without the normal 2-day activation delay.

The default key lifetime is 90 days, though this is configurable as in the following example.

```
services.AddDataProtection()
    // use 14-day lifetime instead of 90-day lifetime
    .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
```

An administrator can also change the default system-wide, though an explicit call to SetDefaultKeyLifetime will override any system-wide policy. The default key lifetime cannot be shorter than 7 days.

Automatic keyring refresh   When the data protection system initializes, it reads the key ring from the underlying repository and caches it in memory. This cache allows Protect and Unprotect operations to proceed without hitting the backing store. The system will automatically check the backing store for changes approximately every 24 hours or when the current default key expires, whichever comes first.

> **:**   Developers should very rarely (if ever) need to use the key management APIs directly. The data protection system will perform automatic key management as described above.

The data protection system exposes an interface IKeyManager that can be used to inspect and make changes to the key ring. The DI system that provided the instance of IDataProtectionProvider can also provide an instance of IKeyManager for your consumption. Alternatively, you can pull the IKeyManager straight from the IServiceProvider as in the example below.

Any operation which modifies the key ring (creating a new key explicitly or performing a revocation) will invalidate the in-memory cache. The next call to Protect or Unprotect will cause the data protection system to reread the key ring and recreate the cache.

The sample below demonstrates using the IKeyManager interface to inspect and manipulate the key ring, including revoking existing keys and generating a new key manually.

```
using System;
using System.IO;
using System.Threading;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // perform a protect operation to force the system to put at least
        // one key in the key ring
        services.GetDataProtector("Sample.KeyManager.v1").Protect("payload");
        Console.WriteLine("Performed a protect operation.");
        Thread.Sleep(2000);

        // get a reference to the key manager
        var keyManager = services.GetService<IKeyManager>();

        // list all keys in the key ring
        var allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked = {key.
        }

        // revoke all keys in the key ring
        keyManager.RevokeAllKeys(DateTimeOffset.Now, reason: "Revocation reason here.");
        Console.WriteLine("Revoked all existing keys.");

        // add a new key to the key ring with immediate activation and a 1-month expiration
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddMonths(1));
        Console.WriteLine("Added a new key.");

        // list all keys in the key ring
        allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked = {key.
```

```
52          }
53      }
54  }
55
56  /*
57   * SAMPLE OUTPUT
58   *
59   * Performed a protect operation.
60   * The key ring contains 1 key(s).
61   * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = False
62   * Revoked all existing keys.
63   * Added a new key.
64   * The key ring contains 2 key(s).
65   * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = True
66   * Key {2266fc40-e2fb-48c6-8ce2-5fde6b1493f7}: Created = 2015-03-18 22:20:51Z, IsRevoked = False
67   */
```

Key storage   The data protection system has a heuristic whereby it tries to deduce an appropriate key storage location and encryption at rest mechanism automatically. This is also configurable by the app developer. The following documents discuss the in-box implementations of these mechanisms:

- *In-box key storage providers*

- *In-box key encryption at rest providers*

Key Storage Providers

By default the data protection system *employs a heuristic* to determine where cryptographic key material should be persisted. The developer can override the heuristic and manually specify the location.

: If you specify an explicit key persistence location, the data protection system will deregister the default key encryption at rest mechanism that the heuristic provided, so keys will no longer be encrypted at rest. It is recommended that you additionally *specify an explicit key encryption mechanism* for production applications.

The data protection system ships with two in-box key storage providers.

File system   We anticipate that the majority of applications will use a file system-based key repository. To configure this, call the PersistKeysToFileSystem configuration routine as demonstrated below, providing a DirectoryInfo pointing to the repository where keys should be stored.

```
sc.AddDataProtection()
    // persist keys to a specific directory
    .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys\"));
```

The DirectoryInfo can point to a directory on the local machine, or it can point to a folder on a network share. If pointing to a directory on the local machine (and the scenario is that only applications on the local machine will need to use this repository), consider using *Windows DPAPI* to encrypt the keys at rest. Otherwise consider using an *X.509 certificate* to encrypt keys at rest.

Registry   Sometimes the application might not have write access to the file system. Consider a scenario where an application is running as a virtual service account (such as w3wp.exe's app pool identity). In these cases, the administrator may have provisioned a registry key that is appropriate ACLed for the service account identity. Call the

---

PersistKeysToRegistry configuration routine as demonstrated below to take advantage of this, providing a RegistryKey pointing to the location where cryptographic key material should be stored.

```
sc.AddDataProtection()
    // persist keys to a specific location in the system registry
    .PersistKeysToRegistry(Registry.CurrentUser.OpenSubKey(@"SOFTWARE\Sample\keys"));
```

If you use the system registry as a persistence mechanism, consider using *Windows DPAPI* to encrypt the keys at rest.

Custom key repository   If the in-box mechanisms are not appropriate, the developer can specify his own key persistence mechanism by providing a custom IXmlRepository.

Key Encryption At Rest

By default the data protection system *employs a heuristic* to determine how cryptographic key material should be encrypted at rest. The developer can override the heuristic and manually specify how keys should be encrypted at rest.

Note: If you specify an explicit key encryption at rest mechanism, the data protection system will deregister the default key storage mechanism that the heuristic provided. You must *specify an explicit key storage mechanism*, otherwise the data protection system will fail to start.   The data protection system ships with three in-box key encryption mechanisms.

Windows DPAPI   *This mechanism is available only on Windows.*

When Windows DPAPI is used, key material will be encrypted via CryptProtectData before being persisted to storage. DPAPI is an appropriate encryption mechanism for data that will never be read outside of the current machine (though it is possible to back these keys up to Active Directory; see DPAPI and Roaming Profiles). For example to configure DPAPI key-at-rest encryption.

```
sc.AddDataProtection()
    // only the local user account can decrypt the keys
    .ProtectKeysWithDpapi();
```

If ProtectKeysWithDpapi is called with no parameters, only the current Windows user account can decipher the persisted key material. You can optionally specify that any user account on the machine (not just the current user account) should be able to decipher the key material, as shown in the below example.

```
sc.AddDataProtection()
    // all user accounts on the machine can decrypt the keys
    .ProtectKeysWithDpapi(protectToLocalMachine: true);
```

X.509 certificate   *This mechanism is not yet available on '.NET Core'.*

If your application is spread across multiple machines, it may be convenient to distribute a shared X.509 certificate across the machines and to configure applications to use this certificate for encryption of keys at rest. See below for an example.

```
sc.AddDataProtection()
    // searches the cert store for the cert with this thumbprint
    .ProtectKeysWithCertificate("3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0");
```

Because this mechanism uses X509Certificate2 and EncryptedXml under the covers, this feature is currently only available on Desktop CLR. Additionally, due to .NET Framework limitations only certificates with CAPI private keys are supported. See *Certificate-based encryption with Windows DPAPI-NG* below for possible workarounds to these limitations.

Windows DPAPI-NG   *This mechanism is available only on Windows 8 / Windows Server 2012 and later.*

Beginning with Windows 8, the operating system supports DPAPI-NG (also called CNG DPAPI). Microsoft lays out its usage scenario as follows.

> Cloud computing, however, often requires that content encrypted on one computer be decrypted on another. Therefore, beginning with Windows 8, Microsoft extended the idea of using a relatively straightforward API to encompass cloud scenarios. This new API, called DPAPI-NG, enables you to securely share secrets (keys, passwords, key material) and messages by protecting them to a set of principals that can be used to unprotect them on different computers after proper authentication and authorization.
>
> From https://msdn.microsoft.com/en-us/library/windows/desktop/hh706794(v=vs.85).aspx

The principal is encoded as a protection descriptor rule. Consider the below example, which encrypts key material such that only the domain-joined user with the specified SID can decrypt the key material.

```
sc.AddDataProtection()
  // uses the descriptor rule "SID=S-1-5-21-..."
  .ProtectKeysWithDpapiNG("SID=S-1-5-21-...",
    flags: DpapiNGProtectionDescriptorFlags.None);
```

There is also a parameterless overload of ProtectKeysWithDpapiNG. This is a convenience method for specifying the rule "SID=mine", where mine is the SID of the current Windows user account.

```
sc.AddDataProtection()
  // uses the descriptor rule "SID={current account SID}"
  .ProtectKeysWithDpapiNG();
```

In this scenario, the AD domain controller is responsible for distributing the encryption keys used by the DPAPI-NG operations. The target user will be able to decipher the encrypted payload from any domain-joined machine (provided that the process is running under their identity).

Certificate-based encryption with Windows DPAPI-NG   If you're running on Windows 8.1 / Windows Server 2012 R2 or later, you can use Windows DPAPI-NG to perform certificate-based encryption, even if the application is running on .NET Core. To take advantage of this, use the rule descriptor string "CERTIFICATE=HashId:thumbprint", where thumbprint is the hex-encoded SHA1 thumbprint of the certificate to use. See below for an example.

```
sc.AddDataProtection()
    // searches the cert store for the cert with this thumbprint
    .ProtectKeysWithDpapiNG("CERTIFICATE=HashId:3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0",
        flags: DpapiNGProtectionDescriptorFlags.None);
```

Any application which is pointed at this repository must be running on Windows 8.1 / Windows Server 2012 R2 or later to be able to decipher this key.

Custom key encryption   If the in-box mechanisms are not appropriate, the developer can specify their own key encryption mechanism by providing a custom IXmlEncryptor.

Key Immutability and Changing Settings

Once an object is persisted to the backing store, its representation is forever fixed. New data can be added to the backing store, but existing data can never be mutated. The primary purpose of this behavior is to prevent data corruption.

One consequence of this behavior is that once a key is written to the backing store, it is immutable. Its creation, activation, and expiration dates can never be changed, though it can revoked by using IKeyManager. Additionally, its underlying algorithmic information, master keying material, and encryption at rest properties are also immutable.

If the developer changes any setting that affects key persistence, those changes will not go into effect until the next time a key is generated, either via an explicit call to IKeyManager.CreateNewKey or via the data protection system's own *automatic key generation* behavior. The settings that affect key persistence are as follows:

- *The default key lifetime*
- *The key encryption at rest mechanism*
- *The algorithmic information contained within the key*

If you need these settings to kick in earlier than the next automatic key rolling time, consider making an explicit call to IKeyManager.CreateNewKey to force the creation of a new key. Remember to provide an explicit activation date ({ now + 2 days } is a good rule of thumb to allow time for the change to propagate) and expiration date in the call.

: All applications touching the repository should specify the same settings with the IDataProtectionBuilder extension methods, otherwise the properties of the persisted key will be dependent on the particular application that invoked the key generation routines.

Key Storage Format

Objects are stored at rest in XML representation. The default directory for key storage is %LOCALAPPDATA%\ASP.NET\DataProtection-Keys\.

The <key> element Keys exist as top-level objects in the key repository. By convention keys have the filename **key-{guid}.xml**, where {guid} is the id of the key. Each such file contains a single key. The format of the file is as follows.

```xml
<?xml version="1.0" encoding="utf-8"?>
<key id="80732141-ec8f-4b80-af9c-c4d2d1ff8901" version="1">
  <creationDate>2015-03-19T23:32:02.3949887Z</creationDate>
  <activationDate>2015-03-19T23:32:02.3839429Z</activationDate>
  <expirationDate>2015-06-17T23:32:02.3839429Z</expirationDate>
  <descriptor deserializerType="{deserializerType}">
    <descriptor>
      <encryption algorithm="AES_256_CBC" />
      <validation algorithm="HMACSHA256" />
      <enc:encryptedSecret decryptorType="{decryptorType}" xmlns:enc="...">
        <encryptedKey>
          <!-- This key is encrypted with Windows DPAPI. -->
          <value>AQAAANCM...8/zeP8lcwAg==</value>
        </encryptedKey>
      </enc:encryptedSecret>
    </descriptor>
  </descriptor>
</key>
```

The <key> element contains the following attributes and child elements:

- The key id. This value is treated as authoritative; the filename is simply a nicety for human readability.

- The version of the <key> element, currently fixed at 1.

- The key's creation, activation, and expiration dates.

- A <descriptor> element, which contains information on the authenticated encryption implementation contained within this key.

In the above example, the key's id is {80732141-ec8f-4b80-af9c-c4d2d1ff8901}, it was created and activated on March 19, 2015, and it has a lifetime of 90 days. (Occasionally the activation date might be slightly before the creation date as in this example. This is due to a nit in how the APIs work and is harmless in practice.)

The <descriptor> element   The outer <descriptor> element contains an attribute deserializerType, which is the assembly-qualified name of a type which implements IAuthenticatedEncryptorDescriptorDeserializer. This type is responsible for reading the inner <descriptor> element and for parsing the information contained within.

The particular format of the <descriptor> element depends on the authenticated encryptor implementation encapsulated by the key, and each deserializer type expects a slightly different format for this. In general, though, this element will contain algorithmic information (names, types, OIDs, or similar) and secret key material. In the above example, the descriptor specifies that this key wraps AES-256-CBC encryption + HMACSHA256 validation.

The <encryptedSecret> element   An <encryptedSecret> element which contains the encrypted form of the secret key material may be present if *encryption of secrets at rest is enabled*. The attribute decryptorType will be the assembly-qualified name of a type which implements IXmlDecryptor. This type is responsible for reading the inner <encryptedKey> element and decrypting it to recover the original plaintext.

As with <descriptor>, the particular format of the <encryptedSecret> element depends on the at-rest encryption mechanism in use. In the above example, the master key is encrypted using Windows DPAPI per the comment.

The <revocation> element   Revocations exist as top-level objects in the key repository. By convention revocations have the filename **revocation-{timestamp}.xml** (for revoking all keys before a specific date) or **revocation-{guid}.xml** (for revoking a specific key). Each file contains a single <revocation> element.

For revocations of individual keys, the file contents will be as below.

```xml
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T22:45:30.2616742Z</revocationDate>
  <key id="eb4fc299-8808-409d-8a34-23fc83d026c9" />
  <reason>human-readable reason</reason>
</revocation>
```

In this case, only the specified key is revoked. If the key id is "*", however, as in the below example, all keys whose creation date is prior to the specified revocation date are revoked.

```xml
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T15:45:45.7366491-07:00</revocationDate>
  <!-- All keys created before the revocation date are revoked. -->
  <key id="*" />
  <reason>human-readable reason</reason>
</revocation>
```

The <reason> element is never read by the system. It is simply a convenient place to store a human-readable reason for revocation.

Ephemeral data protection providers

There are scenarios where an application needs a throwaway IDataProtectionProvider. For example, the developer might just be experimenting in a one-off console application, or the application itself is transient (it's scripted or a unit test project). To support these scenarios the package Microsoft.AspNetCore.DataProtection includes a type EphemeralDataProtectionProvider. This type provides a basic implementation of IDataProtectionProvider whose key repository is held solely in-memory and isn't written out to any backing store.

Each instance of EphemeralDataProtectionProvider uses its own unique master key. Therefore, if an IDataProtector rooted at an EphemeralDataProtectionProvider generates a protected payload, that payload can only be unprotected by an equivalent IDataProtector (given the same *purpose* chain) rooted at the same EphemeralDataProtectionProvider instance.

The following sample demonstrates instantiating an EphemeralDataProtectionProvider and using it to protect and unprotect data.

```
using System;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        const string purpose = "Ephemeral.App.v1";

        // create an ephemeral provider and demonstrate that it can round-trip a payload
        var provider = new EphemeralDataProtectionProvider();
        var protector = provider.CreateProtector(purpose);
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        // if I create a new ephemeral provider, it won't be able to unprotect existing
        // payloads, even if I specify the same purpose
        provider = new EphemeralDataProtectionProvider();
        protector = provider.CreateProtector(purpose);
        unprotectedPayload = protector.Unprotect(protectedPayload); // THROWS
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello!
 * Protect returned: CfDJ8AAAAAAAAAAAAAAAAAAAA...uGoxWLjGKtm1SkNACQ
 * Unprotect returned: Hello!
```

```
 * << throws CryptographicException >>
 */
```

Compatibility

Sharing cookies between applications

Web sites commonly consist of many individual web applications, all working together harmoniously. If an application developer wants to provide a good single-sign-on experience, he'll often need all of the different web applications within the site to share authentication tickets between each other.

To support this scenario, the data protection stack allows sharing Katana cookie authentication and ASP.NET Core cookie authentication tickets.

Sharing authentication cookies between applications To share authentication cookies between two different ASP.NET Core applications, configure each application that should share cookies as follows.

1. Add Authentication to your app

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

2. In your configure method use the CookieAuthenticationOptions to set up the data protection service for cookies

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    DataProtectionProvider = DataProtectionProvider.Create(new DirectoryInfo(@"c:\shared-auth-ticket-
});
```

Caution: When used in this manner, the DirectoryInfo should point to a key storage location specifically set aside for authentication cookies. The application name is ignored (intentionally so, since you're trying to get multiple applications to share payloads). You should consider configuring the DataProtectionProvider such that keys are encrypted at rest, as in the below example.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    DataProtectionProvider = DataProtectionProvider.Create(
        new DirectoryInfo(@"c:\shared-auth-ticket-keys\"),
        configure =>
        {
            configure.ProtectKeysWithCertificate("thumbprint");
        })
});
```

The cookie authentication middleware will use the explicitly provided implementation of the DataProtectionProvider, which due to taking an explicit directory in its constructor is isolated from the data protection system used by other parts of the application.

Sharing authentication cookies between ASP.NET 4.x and ASP.NET Core applications   ASP.NET 4.x applications which use Katana cookie authentication middleware can be configured to generate authentication cookies which are compatible with the ASP.NET Core cookie authentication middleware. This allows upgrading a large site's individual applications piecemeal while still providing a smooth single sign on experience across the site.

Tip: You can tell if your existing application uses Katana cookie authentication middleware by the existence of a call to UseCookieAuthentication in your project's Startup.Auth.cs. ASP.NET 4.x web application projects created with Visual Studio 2013 and later use the Katana cookie authentication middleware by default.

:   Your ASP.NET 4.x application must target .NET Framework 4.5.1 or higher, otherwise the necessary NuGet packages will fail to install.

To share authentication cookies between your ASP.NET 4.x applications and your ASP.NET Core applications, configure the ASP.NET Core application as stated above, then configure your ASP.NET 4.x applications by following the steps below.

1. Install the package Microsoft.Owin.Security.Interop into each of your ASP.NET 4.x applications.

2. In Startup.Auth.cs, locate the call to UseCookieAuthentication, which will generally look like the following.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    // ...
});
```

3. Modify the call to UseCookieAuthentication as follows, changing the AuthenticationType and CookieName to match those of the ASP.NET Core cookie authentication middleware, and providing an instance of a DataProtectionProvider that has been initialized to a key storage location.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationType = "Cookies",
    CookieName = ".AspNetCore.Cookies",
    // CookiePath = "...", (if necessary)
    // ...
    TicketDataFormat = new AspNetTicketDataFormat(
        new DataProtectorShim(
            DataProtectionProvider.Create(new DirectoryInfo(@"c:\shared-auth-ticket-keys\"))
            .CreateProtector("Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationMiddlew
            "Cookies", "v2")))
});

The DirectoryInfo has to point to the same storage location that you pointed your ASP.NET Core applic
```

4. In IdentityModels.cs, change the call to ApplicationUserManager.CreateIdentity to use the same authentication type as in the cookie middleware.

```
public ClaimsIdentity GenerateUserIdentity(ApplicationUserManager manager)
{
    // Note the authenticationType must match the one defined in CookieAuthenticationOptions.Authent
    var userIdentity = manager.CreateIdentity(this, "Cookies");
    // ...
}
```

The ASP.NET 4.x and ASP.NET Core applications are now configured to share authentication cookies.

: You'll need to make sure that the identity system for each application is pointed at the same user database. Otherwise the identity system will produce failures at runtime when it tries to match the information in the authentication cookie against the information in its database.

Replacing <machineKey> in ASP.NET

The implementation of the <machineKey> element in ASP.NET is replaceable. This allows most calls to ASP.NET cryptographic routines to be routed through a replacement data protection mechanism, including the new data protection system.

Package installation

: The new data protection system can only be installed into an existing ASP.NET application targeting .NET 4.5.1 or higher. Installation will fail if the application targets .NET 4.5 or lower.

To install the new data protection system into an existing ASP.NET 4.5.1+ project, install the package Microsoft.AspNetCore.DataProtection.SystemWeb. This will instantiate the data protection system using the *default configuration* settings.

When you install the package, it inserts a line into Web.config that tells ASP.NET to use it for most cryptographic operations, including forms authentication, view state, and calls to MachineKey.Protect. The line that's inserted reads as follows.

```
<machineKey compatibilityMode="Framework45" dataProtectorType="..." />
```

: You can tell if the new data protection system is active by inspecting fields like __VIEWSTATE, which should begin with "CfDJ8" as in the below example. "CfDJ8" is the base64 representation of the magic "09 F0 C9 F0" header that identifies a payload protected by the data protection system.

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="CfDJ8AWPr2EQPTBGs3L2GCZQpk..." />
```

Package configuration   The data protection system is instantiated with a default zero-setup configuration. However, since by default keys are persisted to the local file system, this won't work for applications which are deployed in a farm. To resolve this, you can provide configuration by creating a type which subclasses DataProtectionStartup and overrides its ConfigureServices method.

Below is an example of a custom data protection startup type which configured both where keys are persisted and how they're encrypted at rest. It also overrides the default app isolation policy by providing its own application name.

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.SystemWeb;
using Microsoft.Extensions.DependencyInjection;

namespace DataProtectionDemo
{
    public class MyDataProtectionStartup : DataProtectionStartup
    {
```

```
        public override void ConfigureServices(IServiceCollection services)
        {
            services.AddDataProtection()
                .SetApplicationName("my-app")
                .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\myapp-keys\"))
                .ProtectKeysWithCertificate("thumbprint");
        }
    }
}
```

: You can also use <machineKey applicationName="my-app" ... /> in place of an explicit call to SetApplicationName. This is a convenience mechanism to avoid forcing the developer to create a DataProtectionStartup-derived type if all he wanted to configure was setting the application name.

To enable this custom configuration, go back to Web.config and look for the <appSettings> element that the package install added to the config file. It will look like the below.

```
<appSettings>
  <!--
  If you want to customize the behavior of the ASP.NET Core Data Protection stack, set the
  "aspnet:dataProtectionStartupType" switch below to be the fully-qualified name of a
  type which subclasses Microsoft.AspNetCore.DataProtection.SystemWeb.DataProtectionStartup.
  -->
  <add key="aspnet:dataProtectionStartupType" value="" />
</appSettings>
```

Fill in the blank value with the assembly-qualified name of the DataProtectionStartup-derived type you just created. If the name of the application is DataProtectionDemo, this would look like the below.

```
<add key="aspnet:dataProtectionStartupType"
    value="DataProtectionDemo.MyDataProtectionStartup, DataProtectionDemo" />
```

The newly-configured data protection system is now ready for use inside the application.

1.12.4 Safe storage of app secrets during development

By Rick Anderson, Daniel Roth

This document shows how you can use the Secret Manager tool to keep secrets out of your code. The most important point is you should never store passwords or other sensitive data in source code, and you shouldn't use production secrets in development and test mode. You can instead use the *configuration* system to read these values from environment variables or from values stored using the Secret Manager tool. The Secret Manager tool helps prevent sensitive data from being checked into source control. The *configuration* system can read secrets stored with the Secret Manager tool described in this article.

Sections:

- *Environment variables*
- *Secret Manager*
- *Accessing user secrets via configuration*
- *How the Secret Manager tool works*
- *Additional Resources*

Environment variables

To avoid storing app secrets in code or in local configuration files you store secrets in environment variables. You can setup the *configuration* framework to read values from environment variables by calling `AddEnvironmentVariables`. You can then use environment variables to override configuration values for all previously specified configuration sources.

For example, if you create a new ASP.NET Core web app with individual user accounts, it will add a default connection string to the *appsettings.json* file in the project with the key `DefaultConnection`. The default connection string is setup to use LocalDB, which runs in user mode and doesn't require a password. When you deploy your application to a test or production server you can override the `DefaultConnection` key value with an environment variable setting that contains the connection string (potentially with sensitive credentials) for a test or production database server.

> : Environment variables are generally stored in plain text and are not encrypted. If the machine or process is compromised then environment variables can be accessed by untrusted parties. Additional measures to prevent disclosure of user secrets may still be required.

Secret Manager

The Secret Manager tool provides a more general mechanism to store sensitive data for development work outside of your project tree. The Secret Manager tool is a project tool that can be used to store secrets for a .NET Core project during development. With the Secret Manager tool you can associate app secrets with a specific project and share them across multiple projects.

> : The Secret Manager tool does not encrypt the stored secrets and should not be treated as a trusted store. It is for development purposes only. The keys and values are stored in a JSON configuration file in the user profile directory.

Installing the Secret Manager tool

- Add `SecretManager.Tools` to the `tools` section of the *project.json* file and run `dotnet restore`.

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

- Test the Secret Manager tool by running the following command:

```
dotnet user-secrets -h
```

: When any of the tools are defined in the project.json file, you must be in the same directory in order to use the tooling commands.

The Secret Manager tool will display usage, options and command help.

The Secret Manager tool operates on project specific configuration settings that are stored in your user profile. To use user secrets the project must specify a `userSecretsId` value in its *project.json* file. The value of `userSecretsId` is arbitrary, but is generally unique to the project.

- Add a `userSecretsId` for your project in its *project.json* file:

```
{
"userSecretsId": "aspnet-WebApp1-c23d27a4-eb88-4b18-9b77-2a93f3b15119",

"dependencies": {
```

- Use the Secret Manager tool to set a secret. For example, in a command window from the project directory enter the following:

```
   dotnet user-secrets set MySecret ValueOfMySecret
```

You can run the secret manager tool from other directories, but you must use the `--project` option to pass in the path to the *project.json* file:

```
dotnet user-secrets set MySecret ValueOfMySecret --project c:\work\WebApp1
```

You can also use the Secret Manager tool to list, remove and clear app secrets.

Accessing user secrets via configuration

You access Secret Manager secrets through the configuration system. Add the `Microsoft.Extensions.Configuration.UserSecrets` as a dependency in your *project.json* file and run `dotnet restore`.

```
  "Microsoft.Extensions.Configuration.UserSecrets": "1.0.0-rc2-final",
```

Add the user secrets configuration source to the `Startup` method:

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);

    if (env.IsDevelopment())
    {
        // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID
        builder.AddUserSecrets();
    }

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

You can now access user secrets via the configuration API:

```
string testConfig = Configuration["MySecret"];
```

How the Secret Manager tool works

The secret manager tool abstracts away the implementation details, such as where and how the values are stored. You can use the tool without knowing these implementation details. In the current version, the values are stored in a JSON configuration file in the user profile directory:

- Windows: `%APPDATA%\microsoft\UserSecrets\<userSecretsId>\secrets.json`
- Linux: `~/.microsoft/usersecrets/<userSecretsId>/secrets.json`
- Mac: `~/.microsoft/usersecrets/<userSecretsId>/secrets.json`

The value of `userSecretsId` comes from the value specified in *project.json*.

You should not write code that depends on the location or format of the data saved with the secret manager tool, as these implementation details might change. For example, the secret values are currently *not* encrypted today, but could be someday.

Additional Resources

- *Configuration*.

1.12.5 Enforcing SSL

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.12.6 Anti-Request Forgery

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.12.7 Preventing Open Redirect Attacks

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.12.8 Preventing Cross-Site Scripting

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.12.9 Enabling Cross-Origin Requests (CORS)

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 (CORS) 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. CORS is safer and more flexible than earlier techniques such as JSONP. This topic shows how to enable CORS in your ASP.NET Core application.

Sections:

- *What is "same origin"?*
- *Setting up CORS*
- *Enabling CORS with middleware*
- *Enabling CORS in MVC*
- *CORS policy options*
- *How CORS works*

What is "same origin"?

Two URLs have the same origin if they have identical schemes, hosts, and ports. (RFC 6454)

These two URLs have the same origin:

- http://example.com/foo.html

- http://example.com/bar.html

These URLs have different origins than the previous two:

- http://example.net - Different domain

- http://example.com:9000/foo.html - Different port

- https://example.com/foo.html - Different scheme

- http://www.example.com/foo.html - Different subdomain

: Internet Explorer does not consider the port when comparing origins.

Setting up CORS

To setup CORS for your application you use the `Microsoft.AspNetCore.Cors` package. In your project.json file, add the following:

```
"dependencies": {
  "Microsoft.AspNet.Cors": "6.0.0-rc1-final",
},
```

Add the CORS services in Startup.cs:

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

Enabling CORS with middleware

To enable CORS for your entire application add the CORS middleware to your request pipeline using the `UseCors` extension method. Note that the CORS middleware must precede any defined endpoints in your app that you want to support cross-origin requests (ex. before any call to `UseMvc`).

You can specify a cross-origin policy when adding the CORS middleware using the `CorsPolicyBuilder` class. There are two ways to do this. The first is to call UseCors with a lambda:

```
public void Configure(IApplicationBuilder app)
{
    app.UseCors(builder =>
        builder.WithOrigins("http://example.com"));
}
```

The lambda takes a CorsPolicyBuilder object. I'll describe all of the configuration options later in this topic. In this example, the policy allows cross-origin requests from "http://example.com" and no other origins.

Note that CorsPolicyBuilder has a fluent API, so you can chain method calls:

```
app.UseCors(builder =>
    builder.WithOrigins("http://example.com")
           .AllowAnyHeader()
    );
```

The second approach is to define one or more named CORS policies, and then select the policy by name at run time.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy("AllowSpecificOrigin",
            builder => builder.WithOrigins("http://example.com"));
    });
}

public void Configure(IApplicationBuilder app)
{
    app.UseCors("AllowSpecificOrigin");
    app.Run(async (context) =>
```

```
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

This example adds a CORS policy named "AllowSpecificOrigin". To select the policy, pass the name to UseCors.

Enabling CORS in MVC

You can alternatively use MVC to apply specific CORS per action, per controller, or globally for all controllers. When using MVC to enable CORS the same CORS services are used, but the CORS middleware is not.

Per action

To specify a CORS policy for a specific 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

To specify the CORS policy for a specific controller add the `[EnableCors]` attribute to the controller class. Specify the policy name.

```
[EnableCors("AllowSpecificOrigin")]
public class HomeController : Controller
{
```

Globally

You can enable CORS globally for all controllers by adding 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();
}
```

CORS policy options

This section describes the various options that you can set in a CORS policy.

- *Set the allowed origins*
- *Set the allowed HTTP methods*
- *Set the allowed request headers*
- *Set the exposed response headers*
- *Credentials in cross-origin requests*
- *Set the preflight expiration time*

For some options it may be helpful to read *How CORS works* first.

Set the allowed origins

To allow one or more specific origins:

```
options.AddPolicy("AllowSpecificOrigins",
builder =>
{
    builder.WithOrigins("http://example.com", "http://www.contoso.com");
});
```

To allow all origins:

```
options.AddPolicy("AllowAllOrigins",
    builder =>
    {
        builder.AllowAnyOrigin();
    });
```

Consider carefully before allowing requests from any origin. It means that literally any website can make AJAX calls to your app.

Set the allowed HTTP methods

To specify which HTTP methods are allowed to access the resource.

```
options.AddPolicy("AllowSpecificMethods",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .WithMethods("GET", "POST", "HEAD");
    });
```

To allow all HTTP methods:

```
options.AddPolicy("AllowAllMethods",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowAnyMethod();
    });
```

This affects pre-flight requests and Access-Control-Allow-Methods header.

Set the allowed request headers

A CORS preflight request might include an Access-Control-Request-Headers header, listing the HTTP headers set by the application (the so-called "author request headers").

To whitelist specific headers:

```
options.AddPolicy("AllowHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .WithHeaders("accept", "content-type", "origin", "x-custom-header");
    });
```

To allow all author request headers:

```
options.AddPolicy("AllowAllHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowAnyHeader();
    });
```

Browsers are not entirely consistent in how they set Access-Control-Request-Headers. If you set headers to anything other than "*", you should include at least "accept", "content-type", and "origin", plus any custom headers that you want to support.

Set the exposed response headers

By default, the browser does not expose all of the response headers to the application. (See http://www.w3.org/TR/cors/#simple-response-header.) The response headers that are available by default are:

- Cache-Control
- Content-Language
- Content-Type
- Expires

- Last-Modified

- Pragma

The CORS spec calls these *simple response headers*. To make other headers available to the application:

```
options.AddPolicy("ExposeResponseHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
                .WithExposedHeaders("x-custom-header");
    });
```

Credentials in cross-origin requests

Credentials require special handling in a CORS request. By default, the browser does not send any credentials with a cross-origin request. Credentials include cookies as well as HTTP authentication schemes. To send credentials with a cross-origin request, the client must set XMLHttpRequest.withCredentials to true.

Using XMLHttpRequest directly:

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'http://www.example.com/api/test');
xhr.withCredentials = true;
```

In jQuery:

```
$.ajax({
    type: 'get',
    url: 'http://www.example.com/home',
    xhrFields: {
        withCredentials: true
    }
```

In addition, the server must allow the credentials. To allow cross-origin credentials:

```
options.AddPolicy("AllowCredentials",
    builder =>
    {
        builder.WithOrigins("http://example.com")
                .AllowCredentials();
    });
```

Now the HTTP response will include an Access-Control-Allow-Credentials header, which tells the browser that the server allows credentials for a cross-origin request.

If the browser sends credentials, but the response does not include a valid Access-Control-Allow-Credentials header, the browser will not expose the response to the application, and the AJAX request fails.

Be very careful about allowing cross-origin credentials, because it means a website at another domain can send a logged-in user's credentials to your app on the user's behalf, without the user being aware. The CORS spec also states that setting origins to "*" (all origins) is invalid if the Access-Control-Allow-Credentials header is present.

Set the preflight expiration time

The Access-Control-Max-Age header specifies how long the response to the preflight request can be cached. To set this header:

```
options.AddPolicy("SetPreflightExpiration",
    builder =>
    {
        builder.WithOrigins("http://example.com")
                .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
    });
```

How CORS works

This section describes what happens in a CORS request, at the level of the HTTP messages. It's important to understand how CORS works, so that you can configure the your CORS policy correctly, and troubleshoot if things don't work as you expect.

The CORS specification introduces several new HTTP headers that enable cross-origin requests. If a browser supports CORS, it sets these headers automatically for cross-origin requests; you don't need to do anything special in your JavaScript code.

Here is an example of a cross-origin request. The "Origin" header gives the domain of the site that is making the request:

```
GET http://myservice.azurewebsites.net/api/test HTTP/1.1
Referer: http://myclient.azurewebsites.net/
Accept: */*
Accept-Language: en-US
Origin: http://myclient.azurewebsites.net
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
```

If the server allows the request, it sets the Access-Control-Allow-Origin header. The value of this header either matches the Origin header, or is the wildcard value "*", meaning that any origin is allowed.:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: http://myclient.azurewebsites.net
Date: Wed, 20 May 2015 06:27:30 GMT
Content-Length: 12

Test message
```

If the response does not include the Access-Control-Allow-Origin header, the AJAX request fails. Specifically, the browser disallows the request. Even if the server returns a successful response, the browser does not make the response available to the client application.

Preflight Requests

For some CORS requests, the browser sends an additional request, called a "preflight request", before it sends the actual request for the resource. The browser can skip the preflight request if the following conditions are true:

- The request method is GET, HEAD, or POST, and

- The application does not set any request headers other than Accept, Accept-Language, Content-Language, Content-Type, or Last-Event-ID, and

- The Content-Type header (if set) is one of the following:

    - application/x-www-form-urlencoded

    - multipart/form-data

    - text/plain

The rule about request headers applies to headers that the application sets by calling setRequestHeader on the XML-HttpRequest object. (The CORS specification calls these "author request headers".) The rule does not apply to headers the browser can set, such as User-Agent, Host, or Content-Length.

Here is an example of a preflight request:

```
OPTIONS http://myservice.azurewebsites.net/api/test HTTP/1.1
Accept: */*
Origin: http://myclient.azurewebsites.net
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: accept, x-my-custom-header
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
Content-Length: 0
```

The pre-flight request uses the HTTP OPTIONS method. It includes two special headers:

- Access-Control-Request-Method: The HTTP method that will be used for the actual request.

- Access-Control-Request-Headers: A list of request headers that the application set on the actual request. (Again, this does not include headers that the browser sets.)

Here is an example response, assuming that the server allows the request:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 0
Access-Control-Allow-Origin: http://myclient.azurewebsites.net
Access-Control-Allow-Headers: x-my-custom-header
Access-Control-Allow-Methods: PUT
Date: Wed, 20 May 2015 06:33:22 GMT
```

The response includes an Access-Control-Allow-Methods header that lists the allowed methods, and optionally an Access-Control-Allow-Headers header, which lists the allowed headers. If the preflight request succeeds, the browser sends the actual request, as described earlier.

1.13 Performance

1.13.1 Measuring Application Performance

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.13.2 Caching

In Memory Caching

By Steve Smith

Caching involves keeping a copy of data in a location that can be accessed more quickly than the source data. ASP.NET Core has rich support for caching in a variety of ways, including keeping data in memory on the local server, which is referred to as *in memory caching*.

> **Sections:**
>
> - *Caching Basics*
> - *Configuring In Memory Caching*
> - *Reading and Writing to a Memory Cache*
> - *Cache Dependencies and Callbacks*
> - *Other Resources*

View or download sample code

Caching Basics

Caching can dramatically improve the performance and scalability of ASP.NET applications, by eliminating unnecessary requests to external data sources for data that changes infrequently.

: Caching in all forms (in-memory or distributed, including session state) involves making a copy of data in order to optimize performance. The copied data should be considered ephemeral - it could disappear at any time. Apps should be written to not depend on cached data, but use it when available.

ASP.NET supports several different kinds of caches, the simplest of which is represented by the IMemoryCache interface, which represents a cache stored in the memory of the local web server.

You should always write (and test!) your application such that it can use cached data if it's available, but otherwise will work correctly using the underlying data source.

An in-memory cache is stored in the memory of a single server hosting an ASP.NET app. If an app is hosted by multiple servers in a web farm or cloud hosting environment, the servers may have different values in their local in-memory caches. Apps that will be hosted in server farms or on cloud hosting should use a *distributed cache* to avoid cache consistency problems.

: A common use case for caching is data-driven navigation menus, which rarely change but are frequently read for display within an application. Caching results that do not vary often but which are requested frequently can greatly improve performance by reducing round trips to out of process data stores and unnecessary computation.

Configuring In Memory Caching

To use an in memory cache in your ASP.NET application, add the following dependencies to your *project.json* file:

```
1    "dependencies": {
2      "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0-rc2-final",
3      "Microsoft.AspNetCore.Server.Kestrel": "1.0.0-rc2-final",
4      "Microsoft.Extensions.Caching.Memory": "1.0.0-rc2-final",
5      "Microsoft.Extensions.Logging": "1.0.0-rc2-final",
6      "Microsoft.Extensions.Logging.Console": "1.0.0-rc2-final"
7    },
```

Caching in ASP.NET Core is a *service* that should be referenced from your application by *Dependency Injection*. To register the caching service and make it available within your app, add the following line to your `ConfigureServices` method in `Startup`:

```
1    public void ConfigureServices(IServiceCollection services)
2    {
3        services.AddMemoryCache();
4
```

You utilize caching in your app by requesting an instance of `IMemoryCache` in your controller or middleware constructor. In the sample for this article, we are using a simple middleware component to handle requests by returning customized greeting. The constructor is shown here:

```
1    public GreetingMiddleware(RequestDelegate next,
2        IMemoryCache memoryCache,
3        ILogger<GreetingMiddleware> logger,
4        IGreetingService greetingService)
5    {
6        _next = next;
7        _memoryCache = memoryCache;
8        _greetingService = greetingService;
9        _logger = logger;
10   }
```

Reading and Writing to a Memory Cache

The middleware's `Invoke` method returns the cached data when it's available.

There are two methods for accessing cache entries:

Get `Get` will return the value if it exists, but otherwise returns `null`.

TryGet `TryGet` will assign the cached value to an `out` parameter and return true if the entry exists. Otherwise it returns false.

Use the `Set` method to write to the cache. `Set` accepts the key to use to look up the value, the value to be cached, and a set of `MemoryCacheEntryOptions`. The `MemoryCacheEntryOptions` allow you to specify absolute or sliding time-based cache expiration, caching priority, callbacks, and dependencies. These options are detailed below.

The sample code (shown below) uses the `SetAbsoluteExpiration` method on `MemoryCacheEntryOptions` to cache greetings for one minute.

```
1  public Task Invoke(HttpContext httpContext)
2  {
3      string cacheKey = "GreetingMiddleware-Invoke";
4      string greeting;
5
6      // try to get the cached item; null if not found
7      // greeting = _memoryCache.Get(cacheKey) as string;
8
9      // alternately, TryGet returns true if the cache entry was found
10     if(!_memoryCache.TryGetValue(cacheKey, out greeting))
11     {
12         // fetch the value from the source
13         greeting = _greetingService.Greet("world");
14
15         // store in the cache
16         _memoryCache.Set(cacheKey, greeting,
17             new MemoryCacheEntryOptions()
18             .SetAbsoluteExpiration(TimeSpan.FromMinutes(1)));
19         _logger.LogInformation($"{cacheKey} updated from source.");
20     }
21     else
22     {
23         _logger.LogInformation($"{cacheKey} retrieved from cache.");
24     }
25
26     return httpContext.Response.WriteAsync(greeting);
27 }
28
29
```

In addition to setting an absolute expiration, a sliding expiration can be used to keep frequently requested items in the cache:

```
// keep item in cache as long as it is requested at least
// once every 5 minutes
new MemoryCacheEntryOptions()
  .SetSlidingExpiration(TimeSpan.FromMinutes(5))
```

To avoid having frequently-accessed cache entries growing too stale (because their sliding expiration is constantly reset), you can combine absolute and sliding expirations:

```
// keep item in cache as long as it is requested at least
// once every 5 minutes...
// but in any case make sure to refresh it every hour
new MemoryCacheEntryOptions()
  .SetSlidingExpiration(TimeSpan.FromMinutes(5))
  .SetAbsoluteExpiration(TimeSpan.FromHours(1))
```

By default, an instance of MemoryCache will automatically manage the items stored, removing entries when necessary in response to memory pressure in the app. You can influence the way cache entries are managed by setting their CacheItemPriority when adding the item to the cache. For instance, if you have an item you want to keep in the cache unless you explicitly remove it, you would use the NeverRemove priority option:

```
// keep item in cache indefinitely unless explicitly removed
new MemoryCacheEntryOptions()
  .SetPriority(CacheItemPriority.NeverRemove))
```

When you do want to explicitly remove an item from the cache, you can do so easily using the `Remove` method:

```
cache.Remove(cacheKey);
```

Cache Dependencies and Callbacks

You can configure cache entries to depend on other cache entries, the file system, or programmatic tokens, evicting the entry in response to changes. You can register a callback, which will run when a cache item is evicted.

```
1  {
2      var pause = new ManualResetEvent(false);
3
4      _memoryCache.Set(_cacheKey, _cacheItem,
5          new MemoryCacheEntryOptions()
6          .RegisterPostEvictionCallback(
7              (key, value, reason, substate) =>
8              {
9                  _result = $"'{key}':'{value}' was evicted because: {reason}";
10                 pause.Set();
11             }
12         ));
13
14     _memoryCache.Remove(_cacheKey);
15
16     Assert.True(pause.WaitOne(500));
17
18     Assert.Equal("'key':'value' was evicted because: Removed", _result);
19 }
20
```

The callback is run on a different thread from the code that removes the item from the cache.

> **:** If the callback is used to repopulate the cache it is possible other requests for the cache will take place (and find it empty) before the callback completes, possibly resulting in several threads repopulating the cached value.

Possible eviction reasons are:

None No reason known.

Removed The item was manually removed by a call to `Remove()`

Replaced The item was overwritten.

Expired The item timed out.

TokenExpired The token the item depended upon fired an event.

Capacity The item was removed as part of the cache's memory management process.

You can specify that one or more cache entries depend on a `CancellationTokenSource` by adding the expiration token to the `MemoryCacheEntryOptions` object. When a cached item is invalidated, call `Cancel` on the token, which will expire all of the associated cache entries (with a reason of `TokenExpired`). The following unit test demonstrates this:

```
1  public void CancellationTokenFiresCallback()
2  {
3      var cts = new CancellationTokenSource();
4      var pause = new ManualResetEvent(false);
```

```
5      _memoryCache.Set(_cacheKey, _cacheItem,
6          new MemoryCacheEntryOptions()
7          .AddExpirationToken(new CancellationChangeToken(cts.Token))
8          .RegisterPostEvictionCallback(
9              (key, value, reason, substate) =>
10             {
11                 _result = $"'{key}':'{value}' was evicted because: {reason}";
12                 pause.Set();
13             }
14      ));
15
16      // trigger the token
17      cts.Cancel();
18
19      Assert.True(pause.WaitOne(500));
20
21      Assert.Equal("'key':'value' was evicted because: TokenExpired", _result);
22  }
```

Using a `CancellationTokenSource` allows multiple cache entries to all be expired without the need to create a dependency between cache entries themselves (in which case, you must ensure that the source cache entry exists before it is used as a dependency for other entries).

Cache entries will inherit triggers and timeouts from other entries accessed while creating the new entry. This approach ensures that subordinate cache entries expire at the same time as related entries.

```
1   [Fact]
2   public void CacheEntryDependencies()
3   {
4       var cts = new CancellationTokenSource();
5       var pause = new ManualResetEvent(false);
6
7       using (var cacheEntry = _memoryCache.CreateEntry(_cacheKey))
8       {
9           _memoryCache.Set("master key", "some value",
10              new MemoryCacheEntryOptions()
11              .AddExpirationToken(new CancellationChangeToken(cts.Token)));
12
13          cacheEntry.SetValue(_cacheItem)
14              .RegisterPostEvictionCallback(
15                  (key, value, reason, substate) =>
16                  {
17                      _result = $"'{key}':'{value}' was evicted because: {reason}";
18                      pause.Set();
19                  }
20              );
21      }
22
23      // trigger the token to expire the master item
24      cts.Cancel();
25
26      Assert.True(pause.WaitOne(500));
27
28      Assert.Equal("'key':'value' was evicted because: TokenExpired", _result);
29  }
```

:   When one cache entry is used to create another, the new one copies the existing entry's expiration tokens and time-based expiration settings, if any. It is not expired in response to manual removal or updating of the existing entry.

Other Resources

- *Working with a Distributed Cache*

Working with a Distributed Cache

By Steve Smith

Distributed caches can improve the performance and scalability of ASP.NET Core apps, especially when hosted in a cloud or server farm environment. This article explains how to work with ASP.NET Core's built-in distributed cache abstractions and implementations.

Sections:

- *What is a Distributed Cache*
- *The IDistributedCache Interface*
- *Using a Redis Distributed Cache*
- *Using a SQL Server Distributed Cache*
- *Recommendations*

View or download sample code

What is a Distributed Cache

A distributed cache is shared by multiple app servers (see *Caching Basics*). The information in the cache is not stored in the memory of individual web servers, and the cached data is available to all of the app's servers. This provides several advantages:

1. Cached data is coherent on all web servers. Users don't see different results depending on which web server handles their request

2. Cached data survives web server restarts and deployments. Individual web servers can be removed or added without impacting the cache.

3. The source data store has fewer requests made to it (than with multiple in-memory caches or no cache at all).

: If using a SQL Server Distributed Cache, some of these advantages are only true if a separate database instance is used for the cache than for the app's source data.

Like any cache, a distributed cache can dramatically improve an app's responsiveness, since typically data can be retrieved from the cache much faster than from a relational database (or web service).

Cache configuration is implementation specific. This article describes how to configure both Redis and SQL Server distributed caches. Regardless of which implementation is selected, the app interacts with the cache using a common IDistributedCache interface.

The IDistributedCache Interface

The IDistributedCache interface includes synchronous and asynchronous methods. The interface allows items to be added, retrieved, and removed from the distributed cache implementation. The IDistributedCache interface includes the following methods:

Get, GetAsync Takes a string key and retrieves a cached item as a `byte[]` if found in the cache.

Set, SetAsync Adds an item (as `byte[]`) to the cache using a string key.

Refresh, RefreshAsync Refreshes an item in the cache based on its key, resetting its sliding expiration timeout (if any).

Remove, RemoveAsync Removes a cache entry based on its key.

To use the IDistributedCache interface:

1. Specify the dependencies needed in *project.json*.

2. Configure the specific implementation of IDistributedCache in your `Startup` class's `ConfigureServices` method, and add it to the container there.

3. From the app's *Middleware* or MVC controller classes, request an instance of IDistributedCache from the constructor. The instance will be provided by *Dependency Injection* (DI).

: There is no need to use a Singleton or Scoped lifetime for IDistributedCache instances (at least for the built-in implementations). You can also create an instance wherever you might need one (instead of using *Dependency Injection*), but this can make your code harder to test, and violates the Explicit Dependencies Principle.

The following example shows how to use an instance of IDistributedCache in a simple middleware component:

```
using System.Threading.Tasks;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Http;
using Microsoft.Extensions.Caching.Distributed;
using System.Text;

namespace DistCacheSample
{
    // You may need to install the Microsoft.AspNet.Http.Abstractions package into your project
    public class StartTimeHeader
    {
        private readonly RequestDelegate _next;
        private readonly IDistributedCache _cache;

        public StartTimeHeader(RequestDelegate next,
            IDistributedCache cache)
        {
            _next = next;
            _cache = cache;
        }

        public async Task Invoke(HttpContext httpContext)
        {
            string startTimeString = "Not found.";
            var value = await _cache.GetAsync("lastServerStartTime");
            if (value != null)
            {
```

```
28              startTimeString = Encoding.UTF8.GetString(value);
29          }
30
31          httpContext.Response.Headers.Append("Last-Server-Start-Time", startTimeString);
32
33          await _next.Invoke(httpContext);
34      }
35  }
36
37  // Extension method used to add the middleware to the HTTP request pipeline.
38  public static class StartTimeHeaderExtensions
39  {
40      public static IApplicationBuilder UseStartTimeHeader(this IApplicationBuilder builder)
41      {
42          return builder.UseMiddleware<StartTimeHeader>();
43      }
44  }
45 }
```

In the code above, the cached value is read, but never written. In this sample, the value is only set when a server starts up, and doesn't change. In a multi-server scenario, the most recent server to start will overwrite any previous values that were set by other servers. The `Get` and `Set` methods use the `byte[]` type. Therefore, the string value must be converted using `Encoding.UTF8.GetString` (for `Get`) and `Encoding.UTF8.GetBytes` (for `Set`).

The following code from *Startup.cs* shows the value being set:

```
1          public void Configure(IApplicationBuilder app,
2              IDistributedCache cache)
3          {
4              app.UseIISPlatformHandler();
5
6              var serverStartTimeString = DateTime.Now.ToString();
7              byte[] val = Encoding.UTF8.GetBytes(serverStartTimeString);
8              cache.Set("lastServerStartTime", val);
9
10             app.UseStartTimeHeader();
```

: Since IDistributedCache is configured in the `ConfigureServices` method, it is available to the `Configure` method as a parameter. Adding it as a parameter will allow the configured instance to be provided through DI.

Using a Redis Distributed Cache

Redis is an open source in-memory data store, which is often used as a distributed cache. You can use it locally, and you can configure an Azure Redis Cache for your Azure-hosted ASP.NET Core apps. Your ASP.NET Core app configures the cache implementation using a `RedisDistributedCache` instance.

You configure the Redis implementation in `ConfigureServices` and access it in your app code by requesting an instance of IDistributedCache (see the code above).

In the sample code, a `RedisCache` implementation is used when the server is configured for a `Staging` environment. Thus the `ConfigureStagingServices` method configures the `RedisCache`:

```
1          /// <summary>
2          /// Use Redis Cache in Staging
```

```
3          /// </summary>
4          /// <param name="services"></param>
5          public void ConfigureStagingServices(IServiceCollection services)
6          {
7              // use Redis
8              services.AddSingleton<IDistributedCache>(serviceProvider =>
9                  new RedisCache(new RedisCacheOptions
10                 {
11                     Configuration = "localhost",
12                     InstanceName = "SampleInstance"
13                 }));
14         }
```

: To install Redis on your local machine, install the chocolatey package http://chocolatey.org/packages/redis-64/ and run `redis-server` from a command prompt.

Using a SQL Server Distributed Cache

The SqlServerCache implementation allows the distributed cache to use a SQL Server database as its backing store. The installation script installs a table with the name you specify. The table will have the following schema:

Column Name	Data Type	Allow Nulls
Id	nvarchar(900)	☐
Value	varbinary(MAX)	☐
ExpiresAtTime	datetimeoffset(7)	☐
SlidingExpirationInSecon...	bigint	☑
AbsoluteExpiration	datetimeoffset(7)	☑
		☐

Like all cache implementations, your app should get and set cache values using an instance of IDistributedCache, not a `SqlServerCache`. The sample implements `SqlServerCache` in the `Production` environment (so it is configured in `ConfigureProductionServices`).

```
1          /// <summary>
2          /// Use SQL Server Cache in Production
3          /// </summary>
4          /// <param name="services"></param>
5          public void ConfigureProductionServices(IServiceCollection services)
6          {
7              // Use SQL Server
8              services.AddSingleton<IDistributedCache>(serviceProvider =>
9              new SqlServerCache(new CacheOptions(new SqlServerCacheOptions()
10             {
11                 ConnectionString = @"Data Source=(localdb)\v11.0;Initial Catalog=DistCache;Integrated
12                 SchemaName = "dbo",
13                 TableName = "TestCache"
14             })));
15         }
```

: The `ConnectionString` (and optionally, `SchemaName` and `TableName`) should typically be stored outside
of source control (such as UserSecrets), as they may contain credentials.

Recommendations

When deciding which implementation of IDistributedCache is right for your app, choose between Redis and SQL
Server based on your existing infrastructure and environment, your performance requirements, and your team's expe-
rience. If your team is more comfortable working with Redis, it's an excellent choice. If your team prefers SQL Server,
you can be confident in that implementation as well. Note that A traditional caching solution stores data in-memory
which allows for fast retrieval of data. You should store commonly used data in a cache and store the entire data in a
backend persistent store such as SQL Server or Azure Storage. Redis Cache is a caching solution which gives you high
throughput and low latency as compared to SQL Cache. Also, you should avoid using the in-memory implementation
(`MemoryCache`) in multi-server environments.

Azure Resources:

- Redis Cache on Azure
- SQL Database on Azure

: The in-memory implementation of IDistributedCache should only be used for testing purposes or for applications
that are hosted on just one server instance.

Response Caching

Steve Smith

> **Sections:**
> - *What is Response Caching*
> - *ResponseCache Attribute*

View or download sample code

What is Response Caching

Response caching refers to specifying cache-related headers on HTTP responses made by ASP.NET MVC actions.
These headers specify how you want client and intermediate (proxy) machines to cache responses to certain requests
(if at all). This can reduce the number of requests a client or proxy makes to the web server, since future requests for
the same action may be served from the client or proxy's cache. In this case, the request is never made to the web
server.

First Request

Request: /foo

Client
Browser

Proxy
Server

Web
Server

Response
`cache-control: public`

Request: /foo

Response
`cache-control: public`

Subsequent Request

Request: /foo

Client
Browser

Proxy
Server

Web
Server

Response (from cache)
`cache-control: public`
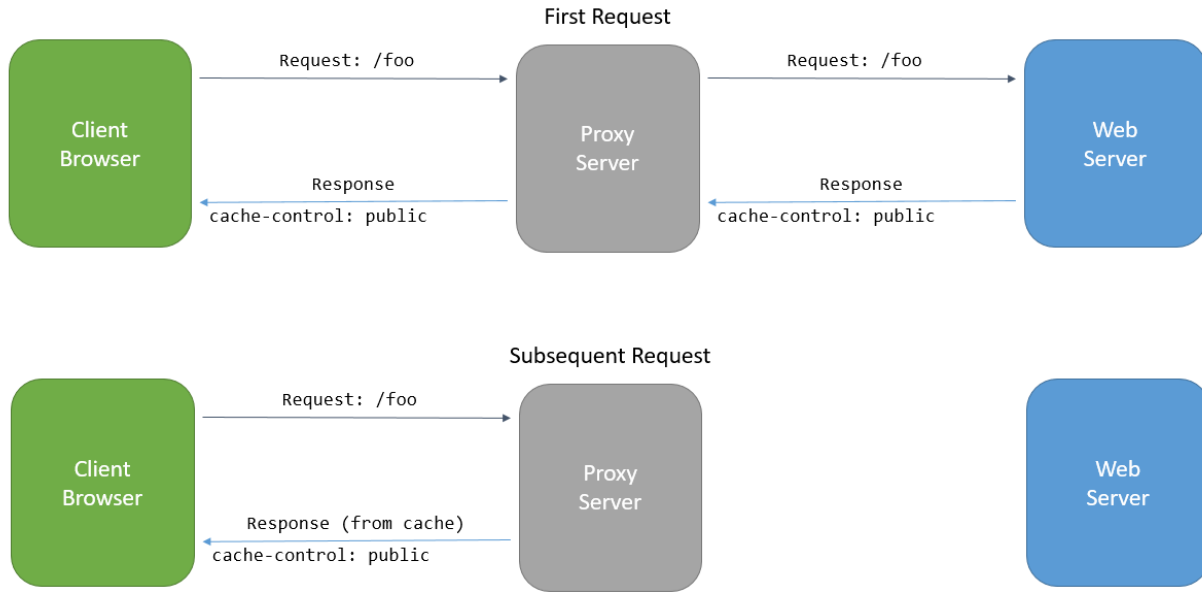
The primary HTTP header used for caching is `Cache-Control`. The HTTP 1.1 specification details many options for this directive. Three common directives are:

public Indicates that the response may be cached.

private Indicates the response is intended for a single user and **must not** be cached by a shared cache. The response could still be cached in a private cache (for instance, by the user's browser).

no-cache Indicates the response **must not** be used by a cache to satisfy any subsequent request (without successful revalidation with the origin server).

: Response caching does not cache responses on the web server. It differs from output caching, which would cache responses in memory on the server in earlier versions of ASP.NET and ASP.NET MVC. Output caching middleware is planned to be added to ASP.NET Core in a future release.

Additional HTTP headers used for caching include `Pragma` and `Vary`, which are described below. Learn more about Caching in HTTP from the specification.

ResponseCache Attribute

The ResponseCacheAttribute is used to specify how a controller action's headers should be set to control its cache behavior. The attribute has the following properties, all of which are optional unless otherwise noted.

Duration `int` The maximum duration (in seconds) the response should be cached. **Required** unless NoStore is `true`.

Location `ResponseCacheLocation` The location where the response may be cached. May be `Any`, `None`, or `Client`. Default is `Any`.

NoStore `bool` Determines whether the value should be stored or not, and overrides other property values. When `true`, `Duration` is ignored and `Location` is ignored for values other than `None`.

VaryByHeader `string` When set, a `vary` response header will be written with the response.

CacheProfileName `string` When set, determines the name of the cache profile to use.

Order `int` The order of the filter (from IOrderedFilter).

The ResponseCacheAttribute is used to configure and create (via IFilterFactory) a ResponseCacheFilter, which performs the work of writing the appropriate HTTP headers to the response. The filter will first remove any existing headers for `Vary`, `Cache-Control`, and `Pragma`, and then will write out the appropriate headers based on the properties set in the ResponseCacheAttribute.

The `Vary` Header This header is only written when the `VaryByHeader` propery is set, in which case it is set to that property's value.

`NoStore` and `Location.None` `NoStore` is a special property that overrides most of the other properties. When this property is true, the `Cache-Control` header will be set to "no-store". Additionally, if `Location` is set to `None`, then `Cache-Control` will be set to "no-store, no-cache" and `Pragma` is likewise set to `no-cache`. (If `NoStore` is false and `Location` is `None`, then both `Cache-Control` and `Pragma` will be set to `no-cache`).

A good scenario in which to set `NoStore` to true is error pages. It's unlikely you would want to respond to a user's request with the error response a different user previously generated, and such responses may include stack traces and other sensitive information that shouldn't be stored on intermdiate servers. For example:

```
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
  return View();
}
```

This will result in the following headers:

```
Cache-Control: no-store,no-cache
Pragma: no-cache
```

Location and Duration To enable caching, `Duration` must be set to a positive value and `Location` must be either `Any` (the default) or `Client`. In this case, the `Cache-Control` header will be set to the location value followed by the "max-age" of the response.

: `Location`'s options of `Any` and `Client` translate into `Cache-Control` header values of `public` and `private`, respectively. As noted previously, setting `Location` to `None` will set both `Cache-Control` and `Pragma` headers to `no-cache`.

Below is an example showing the headers produced by setting `Duration` and leaving the default `Location` value.

```
[ResponseCache(Duration=60)]
```

Produces the following headers:

```
Cache-Control: public,max-age=60
```

Cache Profiles Instead of duplicating `ResponseCache` settings on many controller action attributes, cache profiles can be configured as options when setting up MVC in the `ConfigureServices` method in `Startup`. Values found in a referenced cache profile will be used as the defaults by the `ResponseCache` attribute, and will be overridden by any properties specified on the attribute.

Setting up a cache profile:

```
1   public void ConfigureServices(IServiceCollection services)
2   {
3       services.AddMvc(options =>
4       {
5           options.CacheProfiles.Add("Default",
6               new CacheProfile()
7               {
8                   Duration=60
9               });
10          options.CacheProfiles.Add("Never",
11              new CacheProfile()
12              {
13                  Location = ResponseCacheLocation.None,
14                  NoStore = true
15              });
16      });
17  }
```

Referencing a cache profile:

```
[ResponseCache(CacheProfileName="Default")]
```

: The ResponseCache attribute can be applied both to actions (methods) as well as controllers (classes). Method-level attributes will override the settings specified in class-level attributes.

In the following example, a class-level attribute specifies a duration of 30 while a method-level attributes references a cache profile with a duration set to 60.

```
1   [ResponseCache(Duration=30)]
2   public class HomeController : Controller
3   {
4       [ResponseCache(CacheProfileName = "Default")]
5       public IActionResult Index()
6       {
7           return View();
8       }
```

The resulting header:

```
Cache-Control: public,max-age=60
```

Output Caching

: We are currently working on this topic.

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

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

Learn more about how you can contribute on GitHub.

1.14 Migration

1.14.1 Migrating From ASP.NET MVC to ASP.NET Core MVC

By Rick Anderson, Daniel Roth, Steve Smith and Scott Addie

This article shows how to get started migrating an ASP.NET MVC project to *ASP.NET Core MVC*. In the process, it highlights many of the things that have changed from ASP.NET MVC. Migrating from ASP.NET MVC is a multiple step process and this article covers the initial setup, basic controllers and views, static content, and client-side dependencies. Additional articles cover migrating configuration and identity code found in many ASP.NET MVC projects.

Sections:

- *Create the starter ASP.NET MVC project*
- *Create the ASP.NET Core project*
- *Configure the site to use MVC*
- *Add a controller and view*
- *Controllers and views*
- *Static content*
- *Gulp*
- *NPM*
- *Migrate the layout file*
- *Configure Bundling*
- *Solving HTTP 500 errors*
- *Additional Resources*

Create the starter ASP.NET MVC project

To demonstrate the upgrade, we'll start by creating a ASP.NET MVC app. Create it with the name *WebApp1* so the namespace will match the ASP.NET Core project we create in the next step.
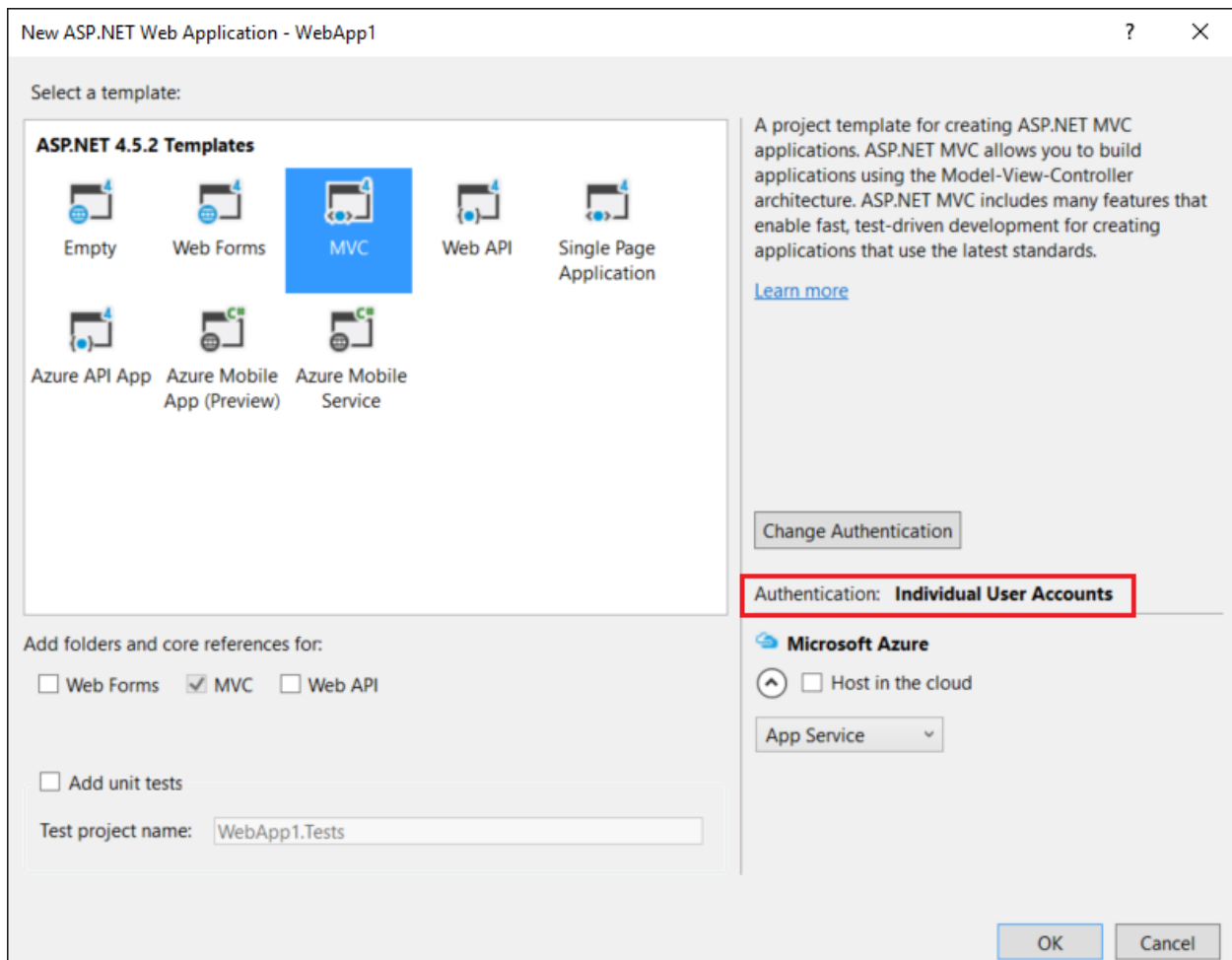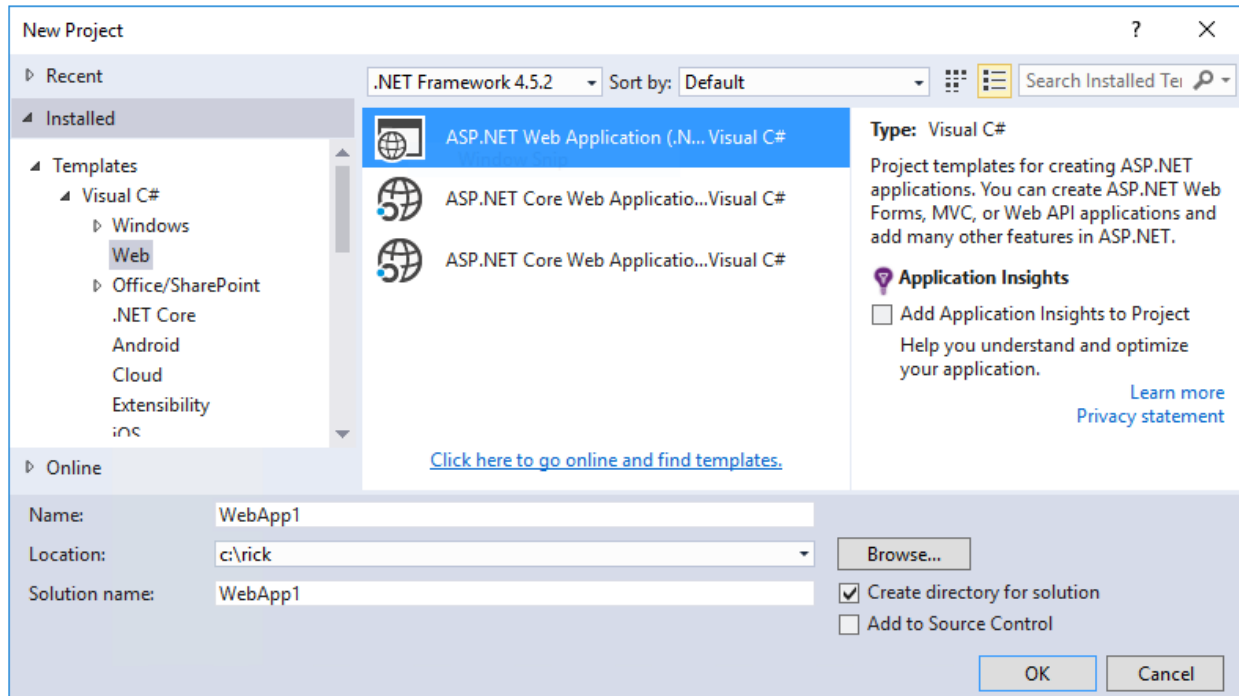
Optional: Change the name of the Solution from *WebApp1* to *Mvc5*. Visual Studio will display the new solution name (*Mvc5*), which will make it easier to tell this project from the next project.

Create the ASP.NET Core project

Create a new *empty* ASP.NET Core web app with the same name as the previous project (*WebApp1*) so the namespaces in the two projects match. Having the same namespace makes it easier to copy code between the two projects. You'll have to create this project in a different directory than the previous project to use the same name.
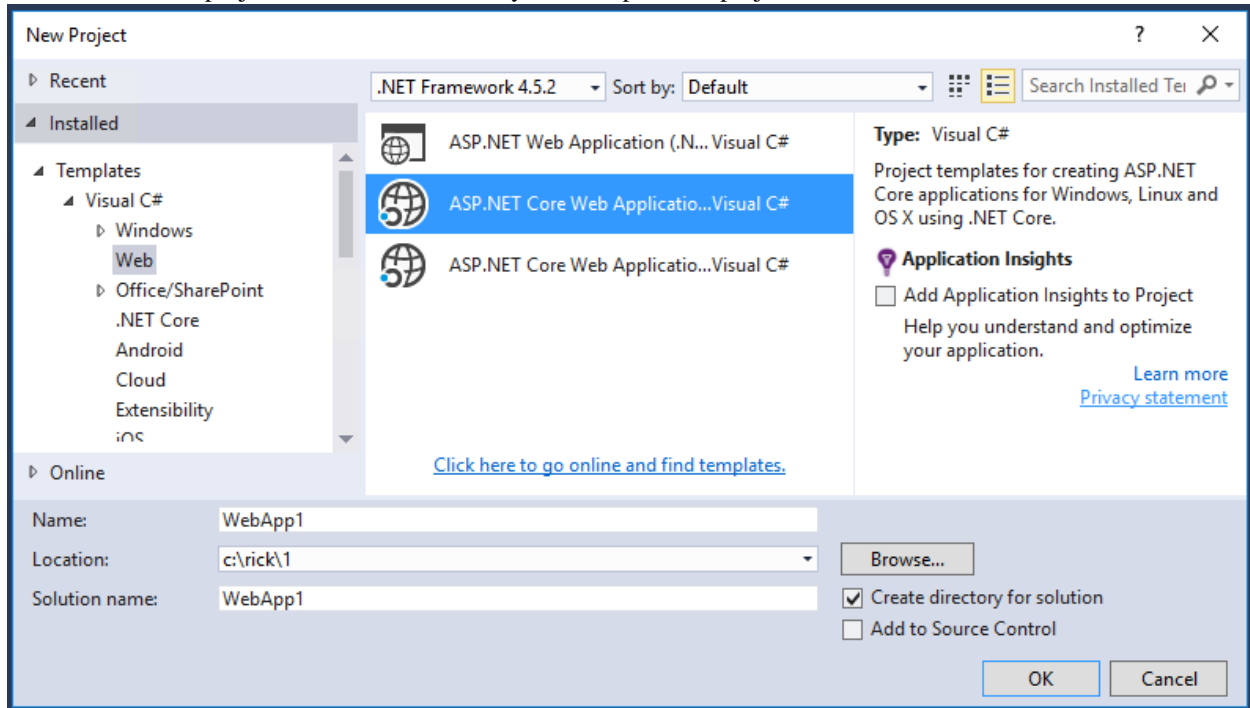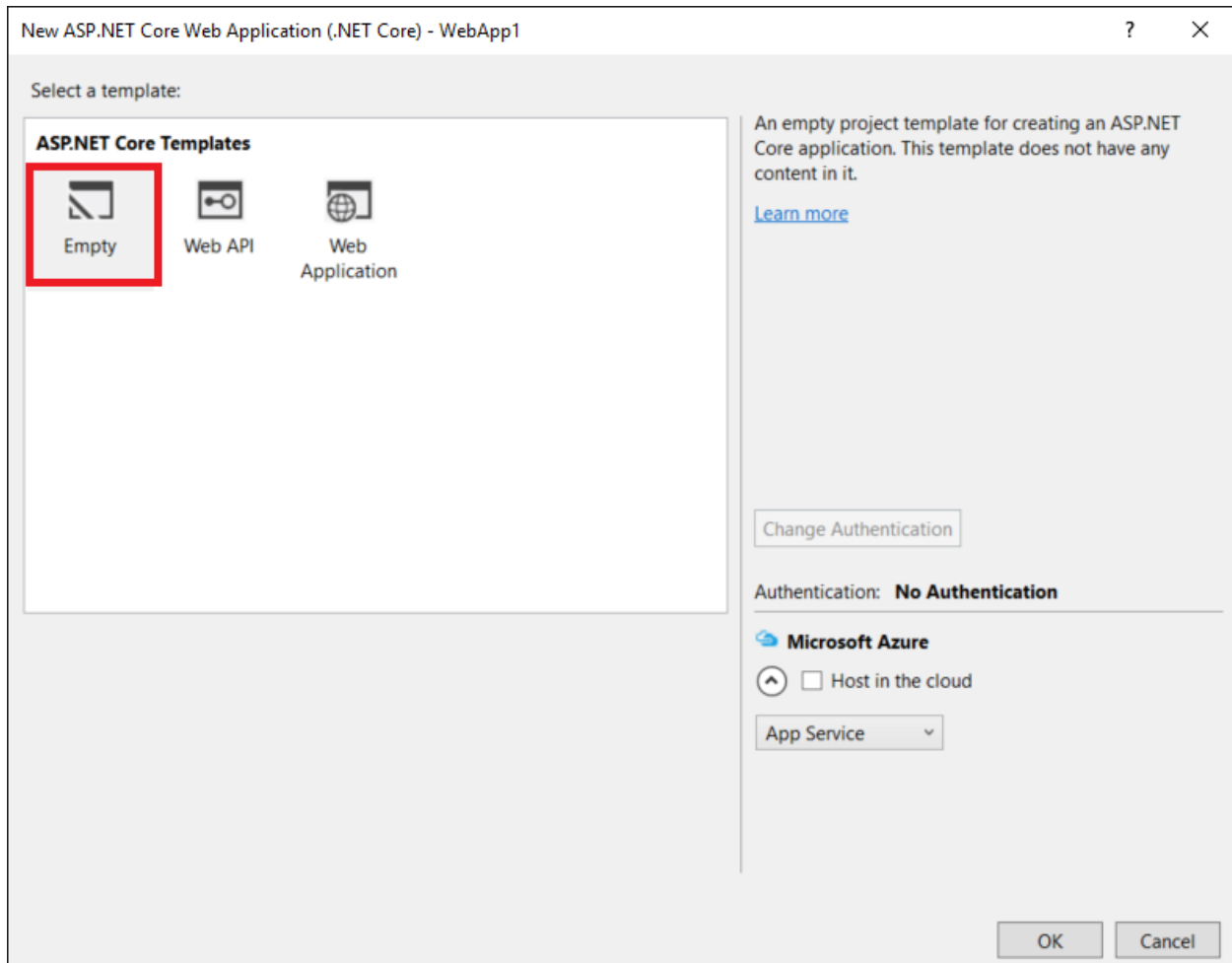
- *Optional:* Create a new ASP.NET Core app named *WebApp1* with authentication set to **Individual User Accounts**. Rename this app *FullAspNetCore*. Creating this project will save you time in the conversion. You can look at the template generated code to see the end result or to copy code to the conversion project. It's also helpful when you get stuck on a conversion step to compare with the template generated project.

Configure the site to use MVC

Open the *project.json* file.

- Add `Microsoft.AspNetCore.Mvc` and `Microsoft.AspNetCore.StaticFiles` to the `dependencies` property:

- Add the "prepublish" line to the "scripts" section:

```
"scripts": {
   "prepublish": [ "npm install", "bower install", "gulp clean", "gulp min" ],
      "postpublish": [ "dotnet publish-iis
         --publish-folder %publish:OutputPath%
         --framework %publish:FullTargetFramework%" ]
}
```

- `Microsoft.AspNetCore.Mvc` Installs in the ASP.NET Core MVC framework package

- `Microsoft.AspNetCore.StaticFiles` is the static file handler. The ASP.NET runtime is modular, and you must explicitly opt in to serve static files (see *Working with Static Files*).

- The "scripts"/"prepublish" line is needed for Gulp, Bower and NPM. We'll talk about that later.

- Open the *Startup.cs* file and change the code to match the following:

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add services to the container.
    // For more information on how to configure your application, visit http://go.microsoft.com/fwlin
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline
    public void Configure(IApplicationBuilder app)
    {
        app.UseStaticFiles();

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

`UseStaticFiles` adds the static file handler. As mentioned previously, the ASP.NET runtime is modular, and you must explicitly opt in to serve static files. `app.UseMvc(routes =>` adds routing. For more information, see *Application Startup* and *Routing*.

Add a controller and view

In this section, you'll add a minimal controller and view to serve as placeholders for the ASP.NET MVC controller and views you'll migrate in the next section.

- Add a *Controllers* folder.

- Add an **MVC controller class** with the name *HomeController.cs* to the *Controllers* folder.
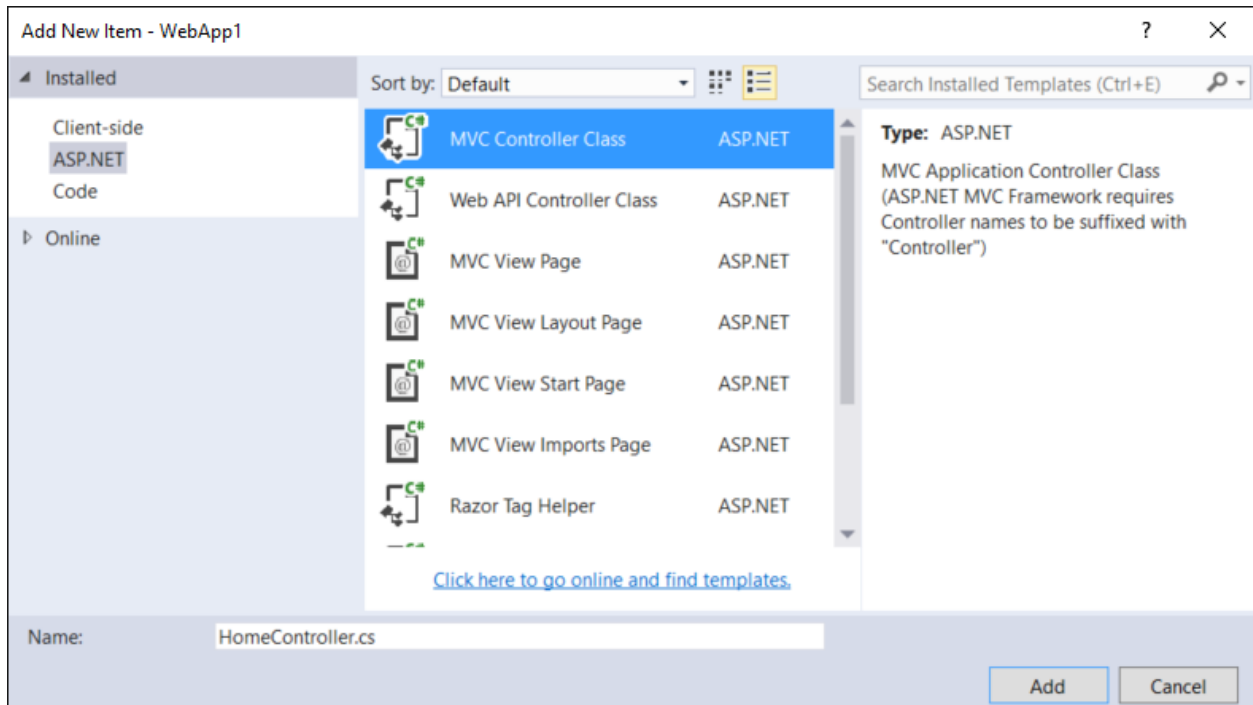
- Add a *Views* folder.

- Add a *Views/Home* folder.

- Add an *Index.cshtml* MVC view page to the *Views/Home* folder.
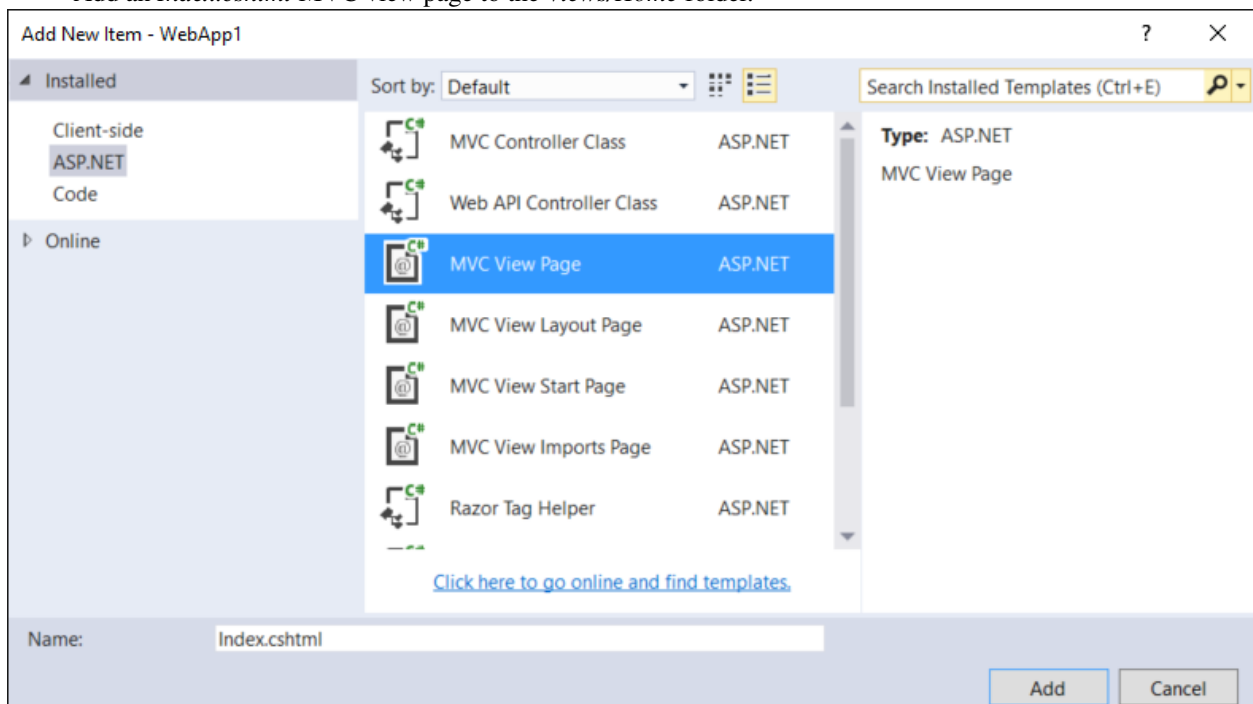
The project structure is shown below:
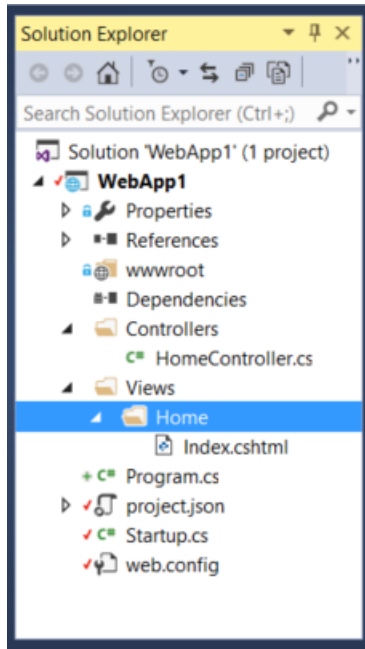
Replace the contents of the *Views/Home/Index.cshtml* file with the following:

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

Run the app.
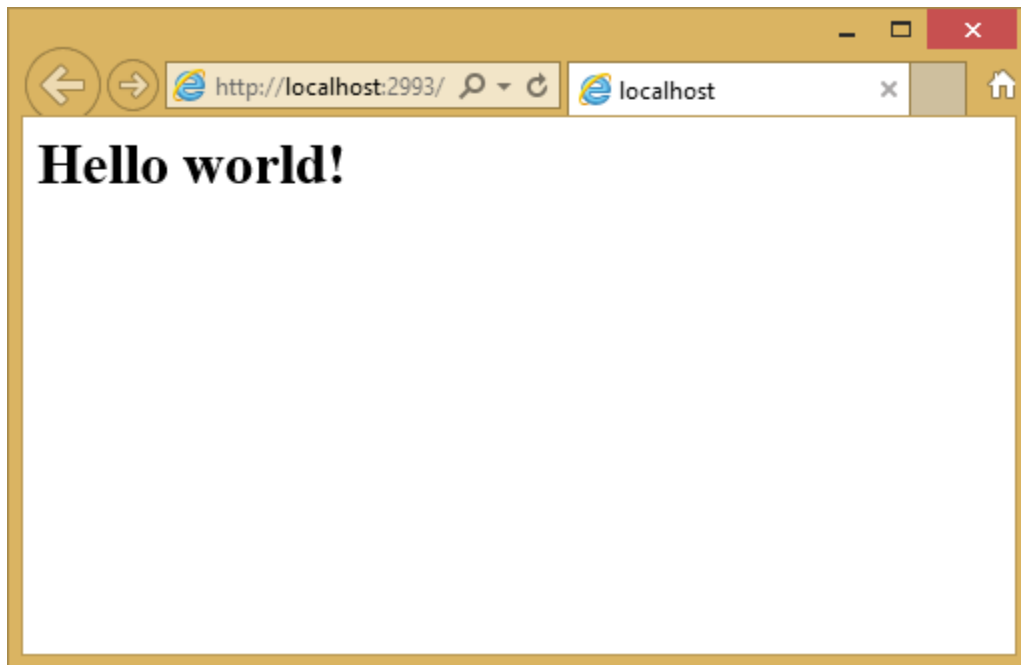
See *Controllers* and *Views* for more information.

Now that we have a minimal working ASP.NET Core project, we can start migrating functionality from the ASP.NET MVC project. We will need to move the following:

- client-side content (CSS, fonts, and scripts)
- controllers

- views

- models

- bundling

- filters

- Log in/out, identity (This will be done in the next tutorial.)

Controllers and views

- Copy each of the methods from the ASP.NET MVC `HomeController` to the new `HomeController`. Note that in ASP.NET MVC, the built-in template's controller action method return type is ActionResult; in ASP.NET Core MVC the actions return IActionResult instead. `ActionResult` implements `IActionResult`, so there is no need to change the return type of your action methods.

- Copy the *About.cshtml*, *Contact.cshtml*, and *Index.cshtml* Razor view files from the ASP.NET MVC project to the ASP.NET Core project.

- Run the ASP.NET Core app and test each method. We haven't migrated the layout file or styles yet, so the rendered views will only contain the content in the view files. You won't have the layout file generated links for the `About` and `Contact` views, so you'll have to invoke them from the browser (replace **2468** with the port number used in your project).

    - http://localhost:2468/home/about

    - http://localhost:2468/home/contact
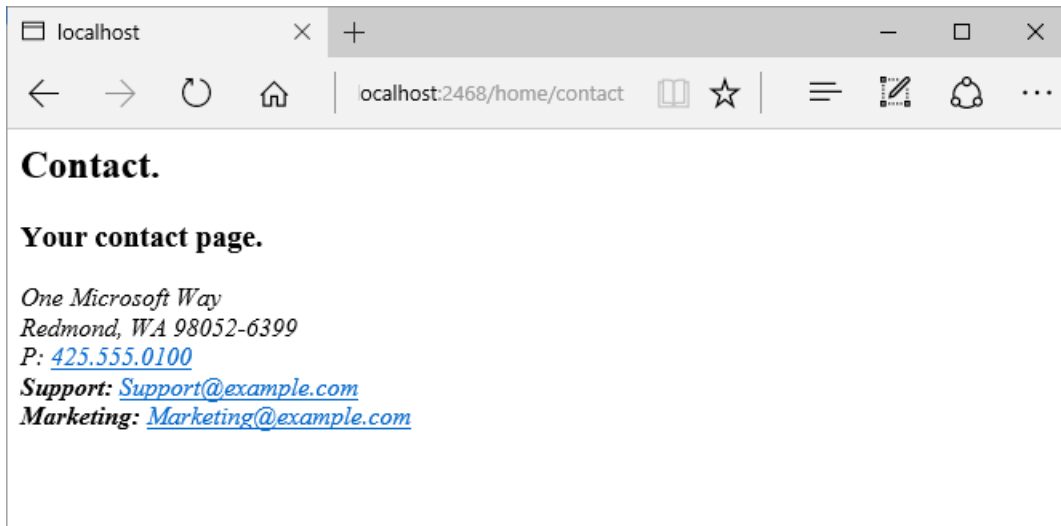
Note the lack of styling and menu items. We'll fix that in the next section.

Static content

In previous versions of ASP.NET MVC, static content was hosted from the root of the web project and was intermixed with server-side files. In ASP.NET Core, static content is hosted in the *wwwroot* folder. You'll want to copy the static content from your old ASP.NET MVC app to the *wwwroot* folder in your ASP.NET Core project. In this sample conversion:

- Copy the *favicon.ico* file from the old MVC project to the *wwwroot* folder in the ASP.NET Core project.

The old ASP.NET MVC project uses Bootstrap for its styling and stores the Bootstrap files in the *Content* and *Scripts* folders. The template-generated old ASP.NET MVC project references Bootstrap in the layout file (*Views/Shared/_Layout.cshtml*). You could copy the *bootstrap.js* and *bootstrap.css* files from the ASP.NET MVC project to the *wwwroot* folder in the new project, but that approach doesn't use the improved mechanism for managing client-side dependencies in ASP.NET Core.

In the new project, we'll add support for Bootstrap (and other client-side libraries) using Bower:

- Add a Bower configuration file named *bower.json* to the project root (Right-click on the project, and then **Add > New Item > Bower Configuration File**). Add Bootstrap and jQuery to the file [1]: (see the highlighted lines below).

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6",
    "jquery": "2.2.0"
  }
}
```

Upon saving the file, Bower will automatically download the dependencies to the *wwwroot/lib* folder. You can use the **Search Solution Explorer** box to find the path of the assets.
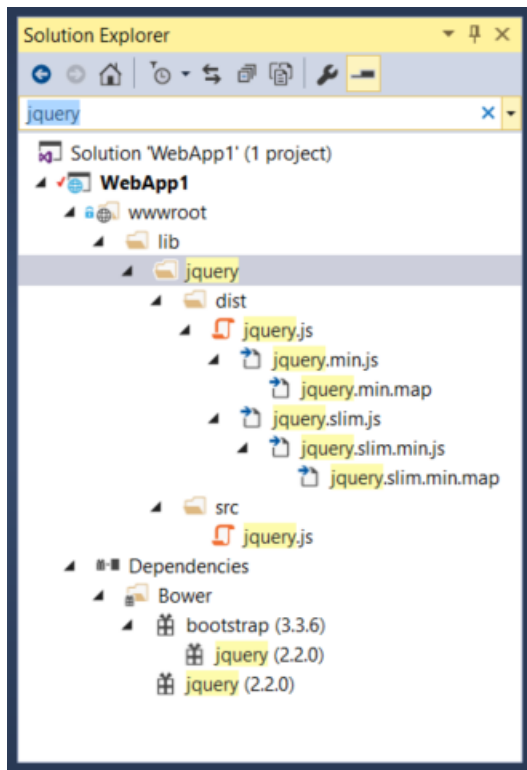
: *bower.json* is not visible in **Solution Explorer**. You can display the hidden *.json* files by selecting the project in **Solution Explorer** and then tapping the **Show All Files** icon. You won't see **Show All Files** unless the project is selected.
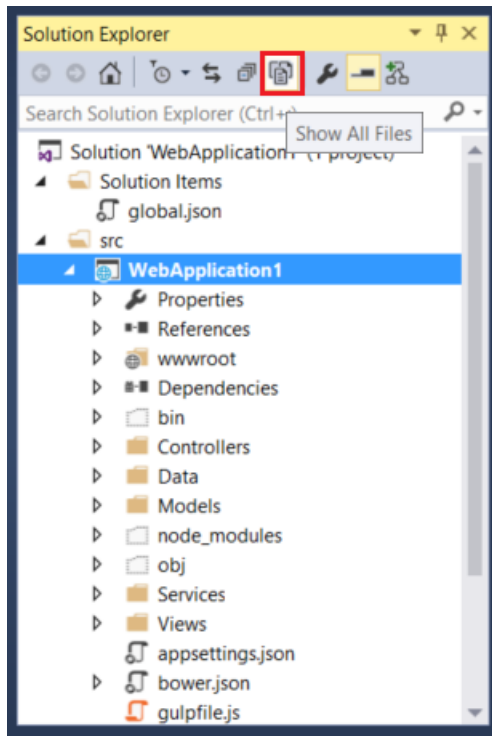
See *Manage Client-Side Packages with Bower* for more information.

Gulp

When you create a new web app using the ASP.NET Core Web Application template, the project is setup to use Gulp. Gulp is a streaming build system for client-side code (HTML, LESS, SASS, etc.). The *gulpfile.js* contains JavaScript that defines a set of gulp tasks that you can set to run automatically on build events or you can run manually using **Task Runner Explorer**. In this section we'll show how to use *gulpfile.js* to bundle and minify the projects JavaScript and CSS files.

If you created the optional *FullAspNetCore* project (a new ASP.NET Core web app with Individual User Accounts), add *gulpfile.js* from that project to the project we are updating. In Solution Explorer, right-click the web app project and choose **Add > Existing Item**.
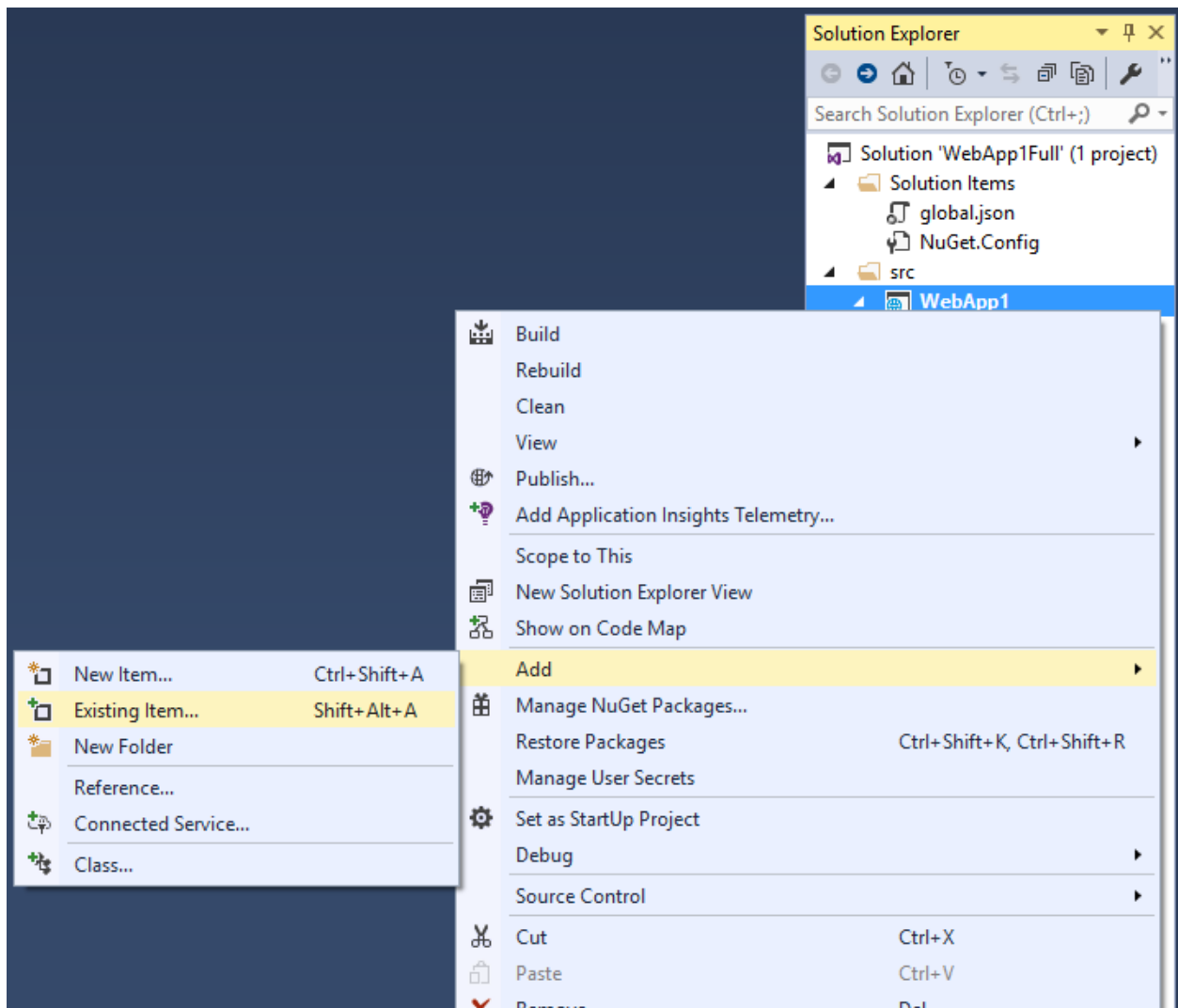
Navigate to *gulpfile.js* from the new ASP.NET Core web app with Individual User Accounts and add the add *gulpfile.js* file. Alternatively, right-click the web app project and choose **Add > New Item**. Select **Gulp Configuration File**, and name the file *gulpfile.js*. The *gulpfile.js* file:

```
/// <binding Clean='clean' />
"use strict";

var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");

var webroot = "./wwwroot/";

var paths = {
    js: webroot + "js/**/*.js",
    minJs: webroot + "js/**/*.min.js",
    css: webroot + "css/**/*.css",
    minCss: webroot + "css/**/*.min.css",
    concatJsDest: webroot + "js/site.min.js",
```

```
    concatCssDest: webroot + "css/site.min.css"
};

gulp.task("clean:js", function (cb) {
    rimraf(paths.concatJsDest, cb);
});

gulp.task("clean:css", function (cb) {
    rimraf(paths.concatCssDest, cb);
});

gulp.task("clean", ["clean:js", "clean:css"]);

gulp.task("min:js", function () {
    return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
    return gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});

gulp.task("min", ["min:js", "min:css"]);
```

The code above performs these functions:

- Cleans (deletes) the target files.

- Minifies the JavaScript and CSS files.

- Bundles (concatenates) the JavaScript and CSS files.

See *Using Gulp*.

NPM

NPM (Node Package Manager) is a package manager which is used to acquire tooling such as Bower and Gulp; and, it is fully supported in Visual Studio. We'll use NPM to manage Gulp dependencies.

If you created the optional *FullAspNetCore* project, add the *package.json* NPM file from that project to the project we are updating. The *package.json* NPM file lists the dependencies for the client-side build processes defined in *gulpfile.js*.

: *package.json* is not visible in **Solution Explorer**.

Right-click the web app project, choose **Add > Existing Item**, and add the *package.json* NPM file. Alternatively, you can add a new NPM configuration file as follows:

1. In Solution Explorer, right-click the project

2. Select **Add > New Item**

3. Select **NPM Configuration File**

4. Leave the default name: *package.json*

5. Tap **Add**

The *package.json* file [1]:

```
{
  "name": "asp.net",
  "version": "0.0.0",
  "private": true,
  "devDependencies": {
    "gulp": "3.8.11",
    "gulp-concat": "2.5.2",
    "gulp-cssmin": "0.1.7",
    "gulp-uglify": "1.2.0",
    "rimraf": "2.2.8"
  }
}
```

Right-click on *gulpfile.js* and select **Task Runner Explorer**. Double-click on a task to run it.

For more information, see *Client-Side Development*.

Migrate the layout file

- Copy the *_ViewStart.cshtml* file from the old ASP.NET MVC project's *Views* folder into the ASP.NET Core project's *Views* folder. The *_ViewStart.cshtml* file has not changed in ASP.NET Core MVC.

- Create a *Views/Shared* folder.

- Optional: Copy *_ViewImports.cshtml* from the old MVC project's *Views* folder into the ASP.NET Core project's *Views* folder. Remove any namespace declaration in the *_ViewImports.cshtml* file. The *_ViewImports.cshtml* file provides namespaces for all the view files and brings in doc:*Tag Helpers </mvc/views/tag-helpers/index>*. *Tag Helpers* are used in the new layout file. The *_ViewImports.cshtml* file is new for ASP.NET Core

- Copy the *_Layout.cshtml* file from the old ASP.NET MVC project's *Views/Shared* folder into the ASP.NET Core project's *Views/Shared* folder.

Open *_Layout.cshtml* file and make the following changes (the completed code is shown below):

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

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

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

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

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

The replacement CSS link:

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

The replacement script tags:

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

The updated _Layout.cshtml file is shown below:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navb
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @*@Html.Partial("_LoginPartial")*@
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

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

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

- Optional: You might want to try using the new layout file. For this project you can copy the layout file from the *FullAspNetCore* project. The new layout file uses *Tag Helpers* and has other improvements.

Configure Bundling

The ASP.NET MVC starter web template utilized the ASP.NET Web Optimization for bundling. In ASP.NET Core, this functionality is performed as part of the build process using Gulp. We've previously configured bundling and minification; all that's left is to change the references to Bootstrap, jQuery and other assets to use the bundled and minified versions. You can see how this is done in the layout file (*Views/Shared/_Layout.cshtml*) of the full template project. See *Bundling and Minification* for more information.

Solving HTTP 500 errors

There are many problems that can cause a HTTP 500 error message that contain no information on the source of the problem. For example, if the *Views/_ViewImports.cshtml* file contains a namespace that doesn't exist in your project, you'll get a HTTP 500 error. To get a detailed error message, add the following code:

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

    app.UseStaticFiles();

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

See **Using the Developer Exception Page** in *Error Handling* for more information.

Additional Resources

- *Client-Side Development*

- *Tag Helpers*

[1] The version numbers in the samples might not be current. You may need to update your projects accordingly.

1.14.2 Migrating Configuration

By Steve Smith and Scott Addie

In the previous article, we began *migrating an ASP.NET MVC project to ASP.NET Core MVC*. In this article, we migrate configuration.

Sections:

- *Setup Configuration*
- *Migrate Configuration Settings from web.config*
- *Summary*

View or download sample code

Setup Configuration

ASP.NET Core no longer uses 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, a *Startup.cs* file was included in the root of the project; and, it was called when

---

the application started. ASP.NET Core has adopted this approach completely placing all startup logic in the *Startup.cs* file.

The *web.config* file has also been replaced in ASP.NET Core. Configuration itself can now be configured, as part of the application startup procedure described in *Startup.cs*. Configuration can still utilize XML files, but typically ASP.NET Core projects will place configuration values in a JSON-formatted file, such as *appsettings.json*. ASP.NET Core'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. See *Configuration* to learn more about configuration in ASP.NET Core.

For this article, we are starting with the partially-migrated ASP.NET Core project from *the previous article*. To setup configuration add the following constructor and property to the *Startup.cs* class located in the root of the project:

```
1   public Startup(IHostingEnvironment env)
2   {
3       // Set up configuration sources.
4       var builder = new ConfigurationBuilder()
5           .AddJsonFile("appsettings.json")
6           .AddEnvironmentVariables();
7       Configuration = builder.Build();
8   }
9
10  public IConfigurationRoot Configuration { get; set; }
```

Note that at this point the *Startup.cs* file will not compile, as we still need to add the following `using` statement:

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

Add an *appsettings.json* file to the root of the project using the appropriate item template:
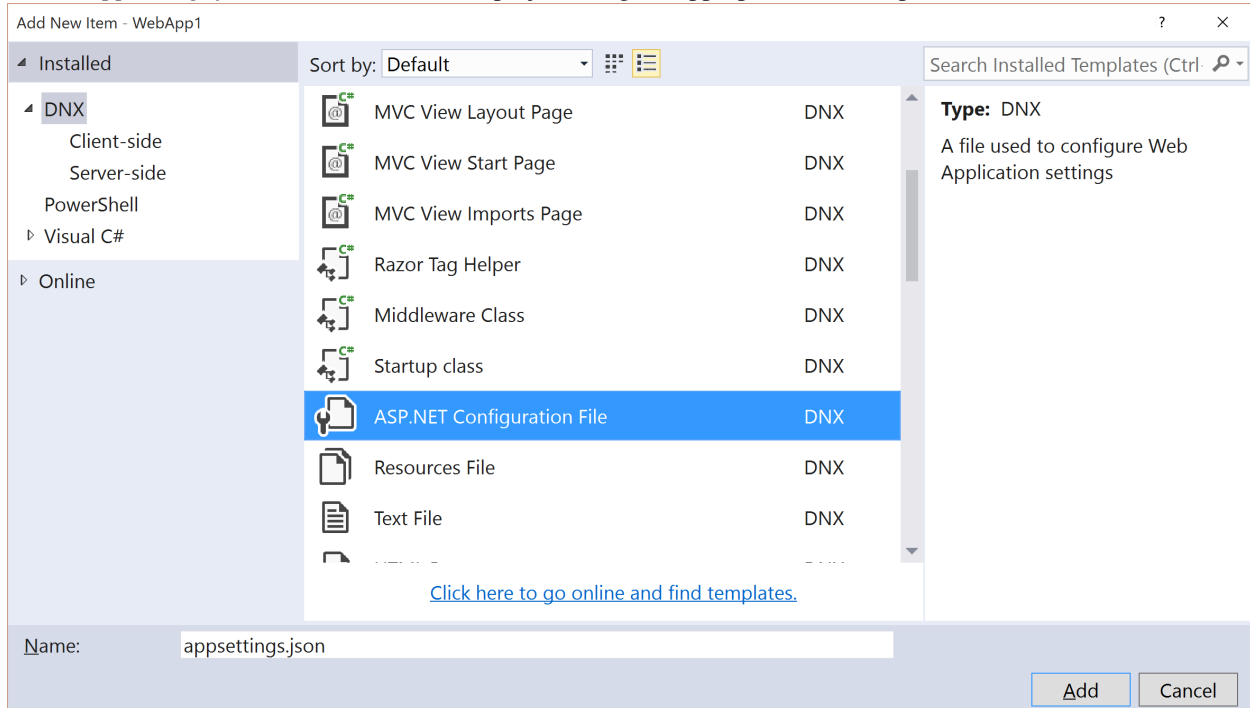
Migrate Configuration Settings from web.config

Our ASP.NET MVC project included the required database connection string in *web.config*, in the
`<connectionStrings>` element. In our ASP.NET Core project, we are going to store this information in the
appsettings.json file. Open *appsettings.json*, and note that it already includes the following:

```
1  {
2          "Data": {
3                  "DefaultConnection": {
4                          "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;Trust
5                  }
6          }
7  }
```

In the highlighted line depicted above, change the name of the database from **_CHANGE_ME** to the name of your
database.

Summary

ASP.NET Core places all startup logic for the application in a single file, in which the necessary services and depen-
dencies 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.

1.14.3 Migrating Authentication and Identity

By Steve Smith

In the previous article we *migrated configuration from an ASP.NET MVC project to ASP.NET Core MVC*. In this article,
we migrate the registration, login, and user management features.

Sections:

- *Configure Identity and Membership*
- *Migrate Registration and Login Logic*
- *Summary*

Configure Identity and Membership

In ASP.NET MVC, authentication and identity features are configured using ASP.NET Identity in Startup.Auth.cs
and IdentityConfig.cs, located in the App_Start folder. In ASP.NET Core MVC, these features are configured in
Startup.cs. Before pulling in the required services and configuring them, we should add the required dependen-
cies to the project. Open *project.json* and add `Microsoft.AspNetCore.Identity.EntityFramework` and
`Microsoft.AspNetCore.Identity.Cookies` to the list of dependencies:

```
"dependencies": {
  "Microsoft.AspNetCore.Mvc": "1.0.0",
  "Microsoft.AspNetCore.Identity.EntityFramework": "1.0.0",
  "Microsoft.AspNetCore.Security.Cookies": "1.0.0"
},
```

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 ASP.NET MVC project: `ApplicationDbContext` and `ApplicationUser`. Create a new *Models* folder in the ASP.NET Core project, and add two classes to it corresponding to these types. You will find the ASP.NET MVC 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.AspNetCore.Identity;

namespace NewMvc6Project.Models
{
  public class ApplicationUser : IdentityUser
  {
  }
}
```

ApplicationDbContext.cs:

```
using Microsoft.AspNetCore.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 database migrations
      // are supported in ASP.NET Core
      if (!_created)
      {
        Database.AsMigrationsEnabled().ApplyMigrations();
        _created = true;
      }
    }

    protected override void OnConfiguring(DbContextOptions options)
    {
      options.UseSqlServer();
    }
```

```
    }
}
```

The ASP.NET Core MVC Starter Web project doesn't include much customization of users, or the ApplicationDb-Context. 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.AspNetCore.Hosting;
using NewMvc6Project.Models;
using Microsoft.AspNetCore.Identity;
```

Our application is now ready to support authentication and identity services - it just needs to have these features exposed to users.

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.

Summary

ASP.NET Core introduces changes to the ASP.NET Identity features. In this article, you have seen how to migrate the authentication and user management features of an ASP.NET Identity to ASP.NET Core.

1.14.4 Migrating from ASP.NET Web API

By Steve Smith and Scott Addie

Web APIs are HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Core MVC includes support for building Web APIs providing a single, consistent way of building web applications. In this article, we demonstrate the steps required to migrate a Web API implementation from ASP.NET Web API to ASP.NET Core MVC.

Sections::

- *Review ASP.NET Web API Project*
- *Create the Destination Project*
- *Migrate Configuration*
- *Migrate Models and Controllers*
- *Summary*

View or download sample code

Review ASP.NET Web API Project

This article uses the sample project, *ProductsApp*, created in the article Getting Started with ASP.NET Web API as its starting point. In that project, a simple ASP.NET Web API project is configured as follows.

In *Global.asax.cs*, a call is made to `WebApiConfig.Register`:

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Web;
5   using System.Web.Http;
6   using System.Web.Routing;
7
8   namespace ProductsApp
9   {
10      public class WebApiApplication : System.Web.HttpApplication
11      {
12          protected void Application_Start()
13          {
14              GlobalConfiguration.Configure(WebApiConfig.Register);
15          }
16      }
17  }
```

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

```csharp
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. It also configures the routing table which is used by ASP.NET Web API. In this case, ASP.NET 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:

```csharp
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)
```

```
25          {
26              var product = products.FirstOrDefault((p) => p.Id == id);
27              if (product == null)
28              {
29                  return NotFound();
30              }
31              return Ok(product);
32          }
33      }
34  }
```

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

```
1   namespace ProductsApp.Models
2   {
3       public class Product
4       {
5           public int Id { get; set; }
6           public string Name { get; set; }
7           public string Category { get; set; }
8           public decimal Price { get; set; }
9       }
10  }
```

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

Create the Destination Project

Using Visual Studio, create a new, empty solution, and name it *WebAPIMigration*. Add the existing *ProductsApp* project to it, then, add a new ASP.NET Core Web Application Project to the solution. Name the new project *ProductsCore*.
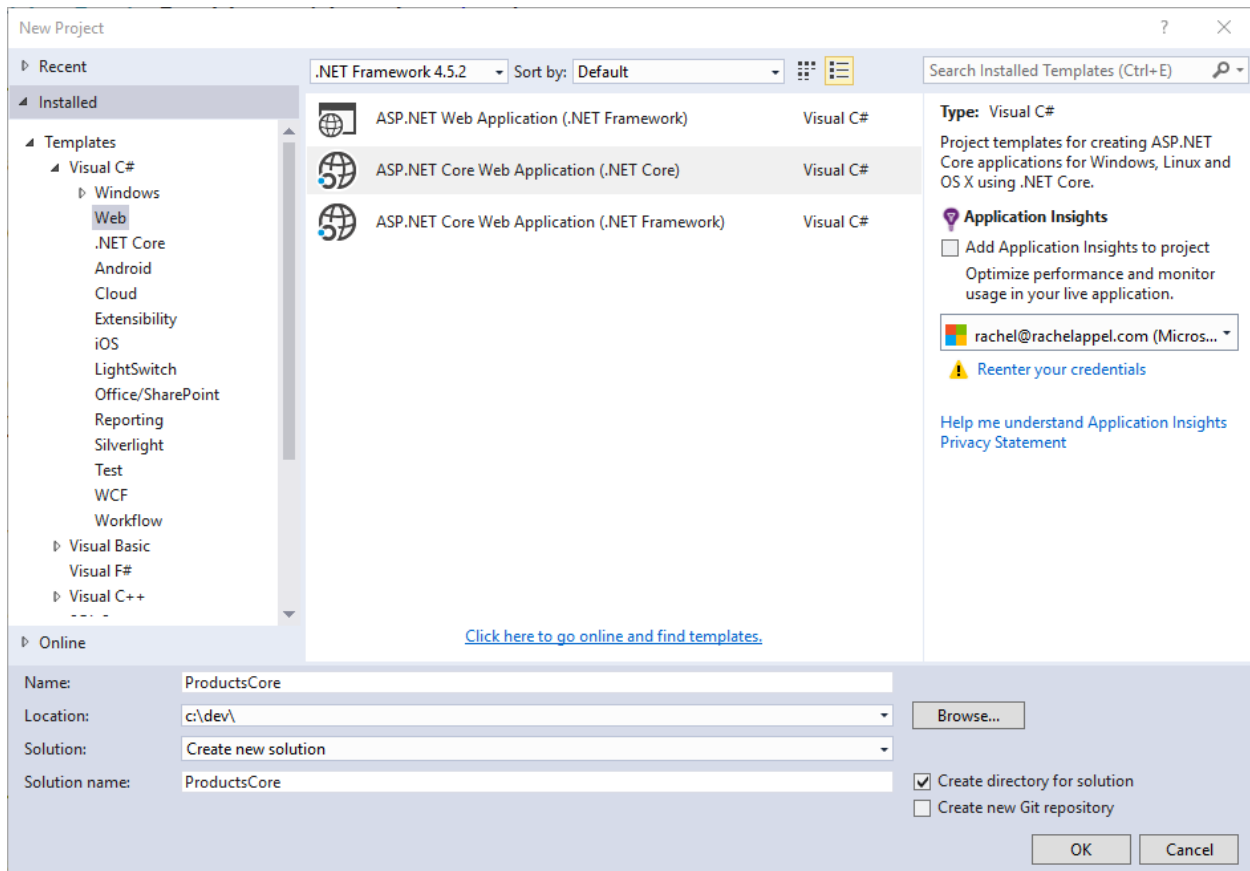
Next, choose the Web API project template. We will migrate the *ProductsApp* contents to this new project.
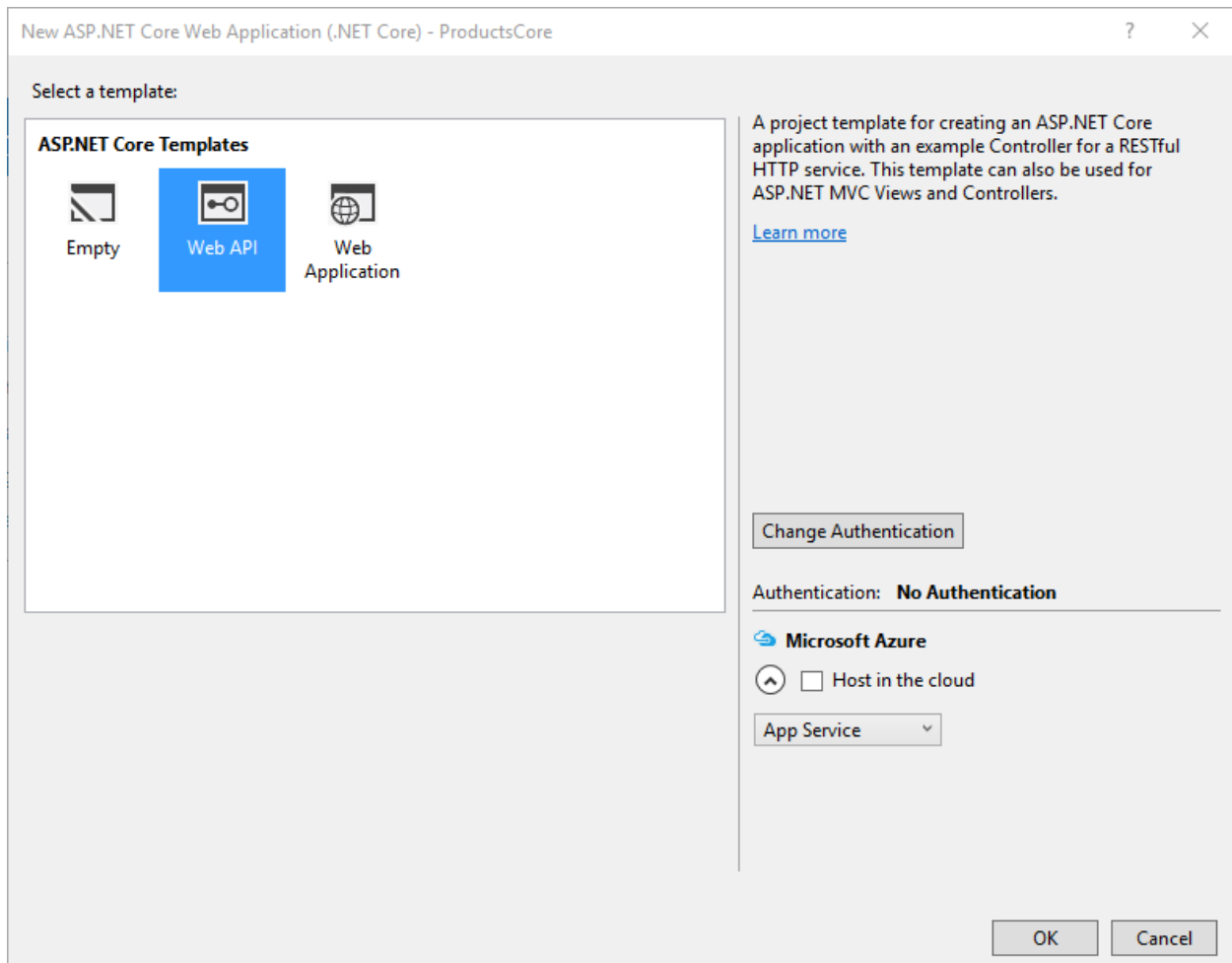
Delete the `Project_Readme.html` file from the new project. Your solution should now look like this:
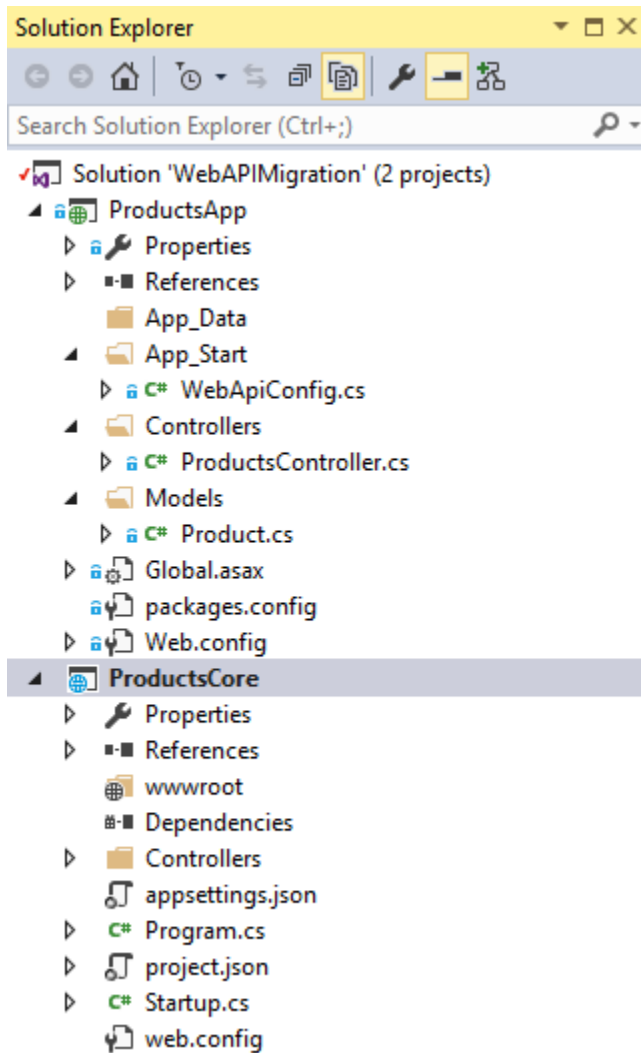
Migrate Configuration

ASP.NET Core 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 (see *Application Startup*). In ASP.NET Core MVC attribute-based routing is now included by default when `UseMvc()` is called and this is the recommended approach for configuring Web API routes (and is how the Web API starter project handles routing).

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace ProductsCore
{
```

```
13    public class Startup
14    {
15        public Startup(IHostingEnvironment env)
16        {
17            // Set up configuration sources.
18            var builder = new ConfigurationBuilder()
19                .AddJsonFile("appsettings.json")
20                .AddEnvironmentVariables();
21            Configuration = builder.Build();
22        }
23
24        public IConfigurationRoot Configuration { get; set; }
25
26        // This method gets called by the runtime. Use this method to add services to the container.
27        public void ConfigureServices(IServiceCollection services)
28        {
29            // Add framework services.
30            services.AddMvc();
31        }
32
33        // This method gets called by the runtime. Use this method to configure the HTTP request pipe
34        public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory logger
35        {
36            loggerFactory.AddConsole(Configuration.GetSection("Logging"));
37            loggerFactory.AddDebug();
38
39            app.UseIISPlatformHandler();
40
41            app.UseStaticFiles();
42
43            app.UseMvc();
44        }
45
46        // Entry point for the application.
47        public static void Main(string[] args) => WebApplication.Run<Startup>(args);
48    }
49 }
```

Assuming you want to use attribute routing in your project going forward, no additional configuration is needed. Simply apply the attributes as needed to your controllers and actions, as is done in the sample `ValuesController` class that is included in the Web API starter project:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Microsoft.AspNet.Mvc;
5
6  namespace ProductsCore.Controllers
7  {
8      [Route("api/[controller]")]
9      public class ValuesController : Controller
10     {
11         // GET: api/values
12         [HttpGet]
13         public IEnumerable<string> Get()
14         {
15             return new string[] { "value1", "value2" };
16         }
```

```
17
18          // GET api/values/5
19          [HttpGet("{id}")]
20          public string Get(int id)
21          {
22              return "value";
23          }
24
25          // POST api/values
26          [HttpPost]
27          public void Post([FromBody]string value)
28          {
29          }
30
31          // PUT api/values/5
32          [HttpPut("{id}")]
33          public void Put(int id, [FromBody]string value)
34          {
35          }
36
37          // DELETE api/values/5
38          [HttpDelete("{id}")]
39          public void Delete(int id)
40          {
41          }
42      }
43  }
```

Note the presence of *[controller]* on line 8. Attribute-based routing now supports certain tokens, such as *[controller]* and *[action]*. These tokens are replaced at runtime with the name of the controller or action, respectively, to which the attribute has been applied. This serves to reduce the number of magic strings in the project, and it ensures the routes will be kept synchronized with their corresponding controllers and actions when automatic rename refactorings are applied.

To migrate the Products API controller, we must first copy *ProductsController* to the new project. Then simply include the route attribute on the controller:

```
[Route("api/[controller]")]
```

You also need to add the `[HttpGet]` attribute to the two methods, since they both should be called via HTTP Get. Include the expectation of an "id" parameter in the attribute for `GetProduct()`:

```
// /api/products
[HttpGet]
...

// /api/products/1
[HttpGet("{id}")]
```

At this point, routing is configured correctly; however, we can't yet test it. Additional changes must be made before *ProductsController* will compile.

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 (*ProductsCore*). At this point, you can build the application, and you will find a number of compilation errors. These should generally fall into the following categories:

- *ApiController* does not exist

- *System.Web.Http* namespace does not exist

- *IHttpActionResult* does not exist

- *NotFound* does not exist

- *Ok* does not exist

Fortunately, these are all very easy to correct:

- Change *ApiController* to *Controller* (you may need to add *using Microsoft.AspNetCore.Mvc*)

- Delete any using statement referring to *System.Web.Http*

- Change any method returning *IHttpActionResult* to return a *IActionResult*

- Change *NotFound* to *HttpNotFound*

- Change *Ok(product)* to *new ObjectResult(product)*

Once these changes have been made and unused using statements removed, the migrated *ProductsController* class looks like this:

```
1   using Microsoft.AspNet.Mvc;
2   using ProductsCore.Models;
3   using System.Collections.Generic;
4   using System.Linq;
5
6   namespace ProductsCore.Controllers
7   {
8       [Route("api/[controller]")]
9       public class ProductsController : Controller
10      {
11          Product[] products = new Product[]
12          {
13              new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
14              new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
15              new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
16          };
17
18          // /api/products
19          [HttpGet]
20          public IEnumerable<Product> GetAllProducts()
21          {
22              return products;
23          }
24
25          // /api/products/1
26          [HttpGet("{id}")]
27          public IActionResult GetProduct(int id)
28          {
29              var product = products.FirstOrDefault((p) => p.Id == id);
30              if (product == null)
31              {
32                  return HttpNotFound();
33              }
34              return new ObjectResult(product);
```

```
35              }
36          }
37  }
```

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

Summary

Migrating a simple ASP.NET Web API project to ASP.NET Core MVC is fairly straightforward, thanks to the built-in support for Web APIs in ASP.NET Core MVC. The main pieces every ASP.NET Web API project will need to migrate are routes, controllers, and models, along with updates to the types used by controllers and actions.

1.14.5 Migrating HTTP Modules to Middleware

By Matt Perdeck

This article shows how to migrate existing ASP.NET HTTP modules and handlers to ASP.NET Core *middleware*.

Sections:

- *Handlers and modules revisited*
- *From handlers and modules to middleware*
- *Migrating module code to middleware*
- *Migrating module insertion into the request pipeline*
- *Migrating handler code to middleware*
- *Migrating handler insertion into the request pipeline*
- *Loading middleware options using the options pattern*
- *Loading middleware options through direct injection*
- *Migrating to the new HttpContext*
- *Additional Resources*

Handlers and modules revisited

Before proceeding to ASP.NET Core middleware, let's first recap how HTTP modules and handlers work:
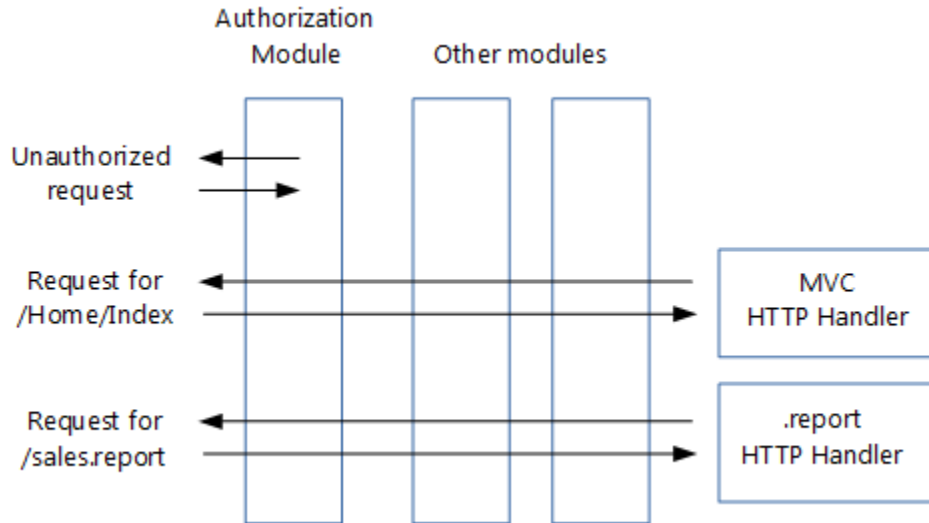
Handlers are:

- Classes that implement IHttpHandler
- Used to handle requests with a given file name or extension, such as *.report*
- Configured in *Web.config*

Modules are:

- Classes that implement IHttpModule
- Invoked for every request
- Able to short-circuit (stop further processing of a request)
- Able to add to the HTTP response, or create their own
- Configured in *Web.config*

The order in which modules process incoming requests is determined by:

1. The application life cycle, which is a series events fired by ASP.NET: BeginRequest, AuthenticateRequest, etc.
   Each module can create a handler for one or more events.

2. For the same event, the order in which they are configured in *Web.config*.

In addition to modules, you can add handlers for the life cycle events to your *Global.asax.cs* file. These handlers run after the handlers in the configured modules.

From handlers and modules to middleware

Middleware are simpler than HTTP modules and handlers:

- Modules, handlers, *Global.asax.cs*, *Web.config* (except for IIS configuration) and the application life cycle are gone
- The roles of both modules and handlers have been taken over by middleware
- Middleware are configured using code rather than in *Web.config*
- *Pipeline branching* lets you send requests to specific middleware, based on not only the URL but also on request headers, query strings, etc.

Middleware are very similar to modules:

- Invoked in principle for every request

- Able to short-circuit a request, by *not passing the request to the next middleware*

- Able to create their own HTTP response

Middleware and modules are processed in a different order:

- Order of middleware is based on the order in which they are inserted into the request pipeline, while order of modules is mainly based on application life cycle events

- Order of middleware for responses is the reverse from that for requests, while order of modules is the same for requests and responses

- See Creating a middleware pipeline with IApplicationBuilder
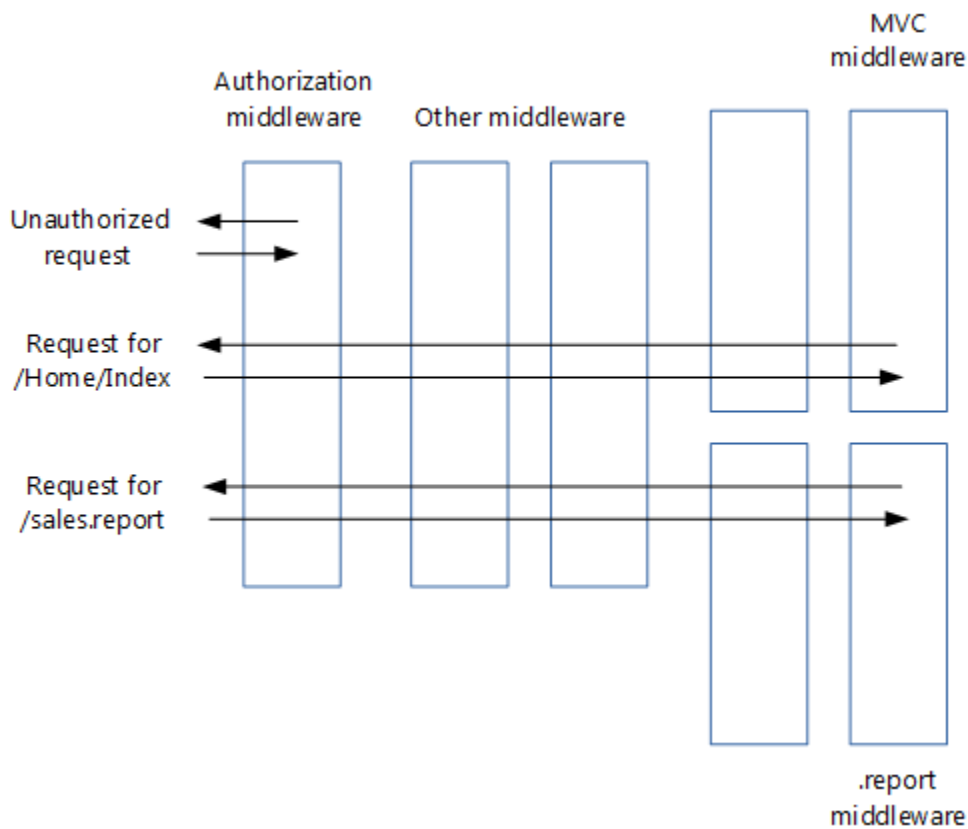
Note how in the image above, the authentication middleware short-circuited the request.

Migrating module code to middleware

An existing HTTP module will look similar to this:

```
1   // ASP.NET 4 module
2
3   using System;
4   using System.Web;
5
```

```
6   namespace MyApp.Modules
7   {
8       public class MyModule : IHttpModule
9       {
10          public void Dispose()
11          {
12          }
13
14          public void Init(HttpApplication application)
15          {
16              application.BeginRequest += (new EventHandler(this.Application_BeginRequest));
17              application.EndRequest += (new EventHandler(this.Application_EndRequest));
18          }
19
20          private void Application_BeginRequest(Object source, EventArgs e)
21          {
22              HttpContext context = ((HttpApplication)source).Context;
23
24              // Do something with context near the beginning of request processing.
25          }
26
27          private void Application_EndRequest(Object source, EventArgs e)
28          {
29              HttpContext context = ((HttpApplication)source).Context;
30
31              // Do something with context near the end of request processing.
32          }
33      }
34  }
```

As shown in the *Middleware* page, an ASP.NET Core middleware is simply a class that exposes an `Invoke` method taking an `HttpContext` and returning a `Task`. Your new middleware will look like this:

```
1   // ASP.NET 5 middleware
2
3   using Microsoft.AspNet.Builder;
4   using Microsoft.AspNet.Http;
5   using System.Threading.Tasks;
6
7   namespace MyApp.Middleware
8   {
9       public class MyMiddleware
10      {
11          private readonly RequestDelegate _next;
12
13          public MyMiddleware(RequestDelegate next)
14          {
15              _next = next;
16          }
17
18          public async Task Invoke(HttpContext context)
19          {
20              // Do something with context near the beginning of request processing.
21
22              await _next.Invoke(context);
23
24              // Clean up.
25          }
```

```
26        }
27
28      public static class MyMiddlewareExtensions
29      {
30          public static IApplicationBuilder UseMyMiddleware(this IApplicationBuilder builder)
31          {
32              return builder.UseMiddleware<MyMiddleware>();
33          }
34      }
35  }
```

The above middleware template was taken from the section on *writing middleware*.

The *MyMiddlewareExtensions* helper class makes it easier to configure your middleware in your `Startup` class. The `UseMyMiddleware` method adds your middleware class to the request pipeline. Services required by the middleware get injected in the middleware's constructor. Your module might terminate a request, for example if the user is not authorized:

```
1   // ASP.NET 4 module that may terminate the request
2
3   private void Application_BeginRequest(Object source, EventArgs e)
4   {
5       HttpContext context = ((HttpApplication)source).Context;
6
7       // Do something with context near the beginning of request processing.
8
9       if (TerminateRequest())
10      {
11          context.Response.End();
12          return;
13      }
14  }
```

A middleware handles this by simply not calling `Invoke` on the next middleware in the pipeline. Keep in mind that this does not fully terminate the request, because previous middlewares will still be invoked when the response makes its way back through the pipeline.

```
1   // ASP.NET 5 middleware that may terminate the request
2
3   public async Task Invoke(HttpContext context)
4   {
5       // Do something with context near the beginning of request processing.
6
7       if (!TerminateRequest())
8           await _next.Invoke(context);
9
10      // Clean up.
11  }
```

When you migrate your module's functionality to your new middleware, you may find that your code doesn't compile because the `HttpContext` class has significantly changed in ASP.NET Core. *Later on*, you'll see how to migrate to the new ASP.NET Core HttpContext.

Migrating module insertion into the request pipeline

HTTP modules are typically added to the request pipeline using *Web.config*:

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <!--ASP.NET 4 web.config-->
3   <configuration>
4       </httpHandlers>
5
6       <modules>
7
8       </handlers>
```

Convert this by adding your new middleware to the request pipeline in your `Startup` class:

```
1   // ASP.NET 5 Startup class
2
3   namespace Asp.Net5
4   {
5       public class Startup
6       {
7           public void Configure(IApplicationBuilder app, IHostingEnvironment env,
8               ILoggerFactory loggerFactory)
9           {
10              // ...
11
12              app.UseMyMiddleware();
13
14              // ...
15          }
16      }
17  }
```

The exact spot in the pipeline where you insert your new middleware depends on the event that it handled as a module (`BeginRequest`, `EndRequest`, etc.) and its order in your list of modules in *Web.config*.

As previously stated, there is no more application life cycle in ASP.NET Core and the order in which responses are processed by middleware differs from the order used by modules. This could make your ordering decision more challenging.

If ordering becomes a problem, you could split your module into multiple middleware that can be ordered independently.

Migrating handler code to middleware

An HTTP handler looks something like this:

```
1   // ASP.NET 4 handler
2
3   using System.Web;
4
5   namespace MyApp.HttpHandlers
6   {
7       public class MyHandler : IHttpHandler
8       {
9           public bool IsReusable { get { return true; } }
10
11          public void ProcessRequest(HttpContext context)
12          {
13              string response = GenerateResponse(context);
```

```
14
15              context.Response.ContentType = GetContentType();
16              context.Response.Output.Write(response);
17          }
18
19          // ...
20      }
21  }
```

In your ASP.NET Core project, you would translate this to a middleware similar to this:

```
1   // ASP.NET 5 middleware migrated from a handler
2
3   using Microsoft.AspNet.Builder;
4   using Microsoft.AspNet.Http;
5   using System.Threading.Tasks;
6
7   namespace MyApp.Middleware
8   {
9       public class MyHandlerMiddleware
10      {
11
12          // Must have constructor with this signature, otherwise exception at run time
13          public MyHandlerMiddleware(RequestDelegate next)
14          {
15              // This is an HTTP Handler, so no need to store next
16          }
17
18          public async Task Invoke(HttpContext context)
19          {
20              string response = GenerateResponse(context);
21
22              context.Response.ContentType = GetContentType();
23              await context.Response.WriteAsync(response);
24          }
25
26          // ...
27      }
28
29      public static class MyHandlerExtensions
30      {
31          public static IApplicationBuilder UseMyHandler(this IApplicationBuilder builder)
32          {
33              return builder.UseMiddleware<MyHandlerMiddleware>();
34          }
35      }
36  }
```

This middleware is very similar to the middleware corresponding to modules. The only real difference is that here there is no call to _next.Invoke(context). That makes sense, because the handler is at the end of the request pipeline, so there will be no next middleware to invoke.

Migrating handler insertion into the request pipeline

Configuring an HTTP handler is done in *Web.config* and looks something like this:

```xml
1  <?xml version="1.0" encoding="utf-8"?>
2  <!--ASP.NET 4 web.config-->
3  <configuration>
4      </httpHandlers>
5
6      </modules>
7
8      <validation validateIntegratedModeConfiguration="false"/>
9      <handlers>
10     </handlers>
```

You could convert this by adding your new handler middleware to the request pipeline in your `Startup` class, similar to middleware converted from modules. The problem with that approach; it would send all requests to your new handler middleware. However, you only want requests with a given extension to reach your middleware. That would give you the same functionality you had with your HTTP handler.

One solution is to branch the pipeline for requests with a given extension, using the `MapWhen` extension method. You do this in the same `Configure` method where you add the other middleware:

```csharp
1  // ASP.NET 5 Startup class
2
3  namespace Asp.Net5
4  {
5      public class Startup
6      {
7          public void Configure(IApplicationBuilder app, IHostingEnvironment env,
8              ILoggerFactory loggerFactory)
9          {
10             // ...
11
12             app.MapWhen(
13                 context => context.Request.Path.ToString().EndsWith(".report"),
14                 appBranch => {
15                     // ... optionally add more middleware to this branch
16                     appBranch.UseMyHandler();
17                 });
18         }
19     }
20 }
```

`MapWhen` takes these parameters:

1. A lambda that takes the `HttpContext` and returns `true` if the request should go down the branch. This means you can branch requests not just based on their extension, but also on request headers, query string parameters, etc.

2. A lambda that takes an `IApplicationBuilder` and adds all the middleware for the branch. This means you can add additional middleware to the branch in front of your handler middleware.

Middleware added to the pipeline before the branch will be invoked on all requests; the branch will have no impact on them.

Loading middleware options using the options pattern

Some modules and handlers have configuration options that are stored in *Web.config*. However, in ASP.NET Core a new configuration model is used in place of *Web.config*.

The new *configuration system* gives you these options to solve this:

- Directly inject the options into the middleware, as shown in the *next section*.

- Use the *options pattern*:

1. Create a class to hold your middleware options, for example:

```
1  public class MyMiddlewareOptions
2  {
3      public string Param1 { get; set; }
4      public string Param2 { get; set; }
5  }
```

2. Store the option values

   The new configuration system allows you to essentially store option values anywhere you want. However, most sites use *appsettings.json*, so we'll take that approach:

```
1  {
2    "MyMiddlewareOptionsSection": {
3      "Param1": "Param1Value",
4      "Param2": "Param2Value"
5    }
6  }
```

   MyMiddlewareOptionsSection here is simply a section name. It doesn't have to be the same as the name of your options class.

3. Associate the option values with the options class

   The options pattern uses ASP.NET Core's dependency injection framework to associate the options type (such as `MyMiddlewareOptions`) with an `MyMiddlewareOptions` object that has the actual options.

   Update your `Startup` class:

   (a) If you're using *appsettings.json*, add it to the configuration builder in the `Startup` constructor:

```
1   public class Startup
2   {
3       public Startup(IHostingEnvironment env)
4       {
5           // Set up configuration sources.
6           var builder = new ConfigurationBuilder()
7               .AddJsonFile("appsettings.json")
8               .AddEnvironmentVariables();
9           Configuration = builder.Build();
10      }
11
12  }
```

   (a) Configure the options service:

```
1   public class Startup
2   {
3       public void ConfigureServices(IServiceCollection services)
4       {
5           services.AddOptions();
6
7           // ...
```

```
8         }
9     }
```

(a) Associate your options with your options class:

```
 1  public class Startup
 2  {
 3      public void ConfigureServices(IServiceCollection services)
 4      {
 5          services.AddOptions();
 6
 7          services.Configure<MyMiddlewareOptions>(
 8              Configuration.GetSection("MyMiddlewareOptionsSection"));
 9
10          // ...
11      }
12  }
```

3. Inject the options into your middleware constructor. This is similar to injecting options into a controller.

```
 1  namespace MyApp.Middleware
 2  {
 3
 4      public class MyMiddlewareWithParams
 5      {
 6          private readonly RequestDelegate _next;
 7          private readonly MyMiddlewareOptions _myMiddlewareOptions;
 8
 9          public MyMiddlewareWithParams(RequestDelegate next,
10              IOptions<MyMiddlewareOptions> optionsAccessor)
11          {
12              _next = next;
13              _myMiddlewareOptions = optionsAccessor.Value;
14          }
15
16          public async Task Invoke(HttpContext context)
17          {
18              // Do something with context near the beginning of request processing
19              // using configuration in _myMiddlewareOptions
20
21              await _next.Invoke(context);
22
23              // Do something with context near the end of request processing
24              // using configuration in _myMiddlewareOptions
25          }
26      }
```

The *UseMiddleware* extension method that adds your middleware to the `IApplicationBuilder` takes care of dependency injection.

This is not limited to `IOptions` objects. Any other object that your middleware requires can be injected this way.

Loading middleware options through direct injection

The options pattern has the advantage that it creates loose coupling between options values and their consumers. Once you've associated an options class with the actual options values, any other class can get access to the options through

the dependency injection framework. There is no need to pass around options values.

This breaks down though if you want to use the same middleware twice, with different options. For example an authorization middleware used in different branches allowing different roles. You can't associate two different options objects with the one options class.

The solution is to get the options objects with the actual options values in your `Startup` class and pass those directly to each instance of your middleware.

1. Add a second key to *appsettings.json*

   To add a second set of options to the *appsettings.json* file, simply use a new key to uniquely identify it:

```
1   {
2     "MyMiddlewareOptionsSection2": {
3       "Param1": "Param1Value2",
4       "Param2": "Param2Value2"
5     },
6     "MyMiddlewareOptionsSection": {
7       "Param1": "Param1Value",
8       "Param2": "Param2Value"
9     }
10  }
```

2. Retrieve options values. The `Get` method on the `Configuration` property lets you retrieve options values:

```
1   // ASP.NET 5 Startup class
2
3   namespace Asp.Net5
4   {
5       public class Startup
6       {
7           public void Configure(IApplicationBuilder app, IHostingEnvironment env,
8               ILoggerFactory loggerFactory)
9           {
10              // ...
11
12              var myMiddlewareOptions =
13                  Configuration.Get<MyMiddlewareOptions>("MyMiddlewareOptionsSection");
14
15              var myMiddlewareOptions2 =
16                  Configuration.Get<MyMiddlewareOptions>("MyMiddlewareOptionsSection2");
17
18              // ...
19
20          }
21      }
22  }
```

3. Pass options values to middleware. The `Use...` extension method (which adds your middleware to the pipeline) is a logical place to pass in the option values:

```
1   // ASP.NET 5 Startup class
2
3   namespace Asp.Net5
4   {
5       public class Startup
6       {
```

```
7          public void Configure(IApplicationBuilder app, IHostingEnvironment env,
8              ILoggerFactory loggerFactory)
9          {
10             // ...
11
12             var myMiddlewareOptions =
13                 Configuration.Get<MyMiddlewareOptions>("MyMiddlewareOptionsSection");
14
15             var myMiddlewareOptions2 =
16                 Configuration.Get<MyMiddlewareOptions>("MyMiddlewareOptionsSection2");
17
18             app.UseMyMiddlewareWithParams(myMiddlewareOptions);
19
20             // ...
21
22             app.UseMyMiddlewareWithParams(myMiddlewareOptions2);
23         }
24     }
25 }
```

4. Enable middleware to take an options parameter. Provide an overload of the `Use...` extension method (that takes the options parameter and passes it to `UseMiddleware`). When `UseMiddleware` is called with parameters, it passes the parameters to your middleware constructor when it instantiates the middleware object.

```
1  using Microsoft.AspNet.Builder;
2  using Microsoft.AspNet.Http;
3  using Microsoft.Extensions.OptionsModel;
4  using System.Threading.Tasks;
5
6  namespace MyApp.Middleware
7  {
8
9      public static class MyMiddlewareWithParamsExtensions
10     {
11         public static IApplicationBuilder UseMyMiddlewareWithParams(
12             this IApplicationBuilder builder)
13         {
14             return builder.UseMiddleware<MyMiddlewareWithParams>();
15         }
16
17         public static IApplicationBuilder UseMyMiddlewareWithParams(
18             this IApplicationBuilder builder, MyMiddlewareOptions myMiddlewareOptions)
19         {
20             return builder.UseMiddleware<MyMiddlewareWithParams>(
21                 new OptionsWrapper<MyMiddlewareOptions>(myMiddlewareOptions));
22         }
23     }
24 }
```

Note how this wraps the options object in an `OptionsWrapper` object. This implements `IOptions`, as expected by the middleware constructor:

```
1      // Remove this when Microsoft.Extensions.Options becomes available from NuGet
2      public class OptionsWrapper<TOptions> : IOptions<TOptions> where TOptions : class, new()
3      {
4          public OptionsWrapper(TOptions options)
5          {
```

```
6                  Value = options;
7              }
8
9          public TOptions Value { get; }
10         }
```

Migrating to the new HttpContext

You saw earlier that the `Invoke` method in your middleware takes a parameter of type `HttpContext`:

```
public async Task Invoke(HttpContext context)
```

`HttpContext` has significantly changed in ASP.NET Core. This section shows how to translate the most commonly used properties of System.Web.HttpContext to the new Microsoft.AspNetCore.Http.HttpContext.

HttpContext

HttpContext.Items translates to:

```
IDictionary<object, object> items = httpContext.Items;
```

Unique request ID (no System.Web.HttpContext counterpart)

Gives you a unique id for each request. Very useful to include in your logs.

```
string requestId = httpContext.TraceIdentifier;
```

HttpContext.Request

HttpContext.Request.HttpMethod translates to:

```
string httpMethod = httpContext.Request.Method;
```

HttpContext.Request.QueryString translates to:

```
IReadableStringCollection queryParameters = httpContext.Request.Query;

// If no query parameter "key" used, values will have 0 items
// If single value used for a key (...?key=v1), values will have 1 item ("v1")
// If key has multiple values (...?key=v1&key=v2), values will have 2 items ("v1" and "v2")
IList<string> values = queryParameters["key"];

// If no query parameter "key" used, value will be ""
// If single value used for a key (...?key=v1), value will be "v1"
// If key has multiple values (...?key=v1&key=v2), value will be "v1,v2"
string value = queryParameters["key"].ToString();
```

HttpContext.Request.Url and HttpContext.Request.RawUrl translate to:

```
// using Microsoft.AspNet.Http.Extensions;
var url = httpContext.Request.GetDisplayUrl();
```

HttpContext.Request.IsSecureConnection translates to:

```
var isSecureConnection = httpContext.Request.IsHttps;
```

HttpContext.Request.UserHostAddress translates to:

```
var userHostAddress = httpContext.Connection.RemoteIpAddress?.ToString();
```

HttpContext.Request.Cookies translates to:

```
IReadableStringCollection cookies = httpContext.Request.Cookies;
string unknownCookieValue = cookies["unknownCookie"]; // will be null (no exception)
string knownCookieValue = cookies["cookie1name"];     // will be actual value
```

HttpContext.Request.Headers translates to:

```
// using Microsoft.AspNet.Http.Headers;
// using Microsoft.Net.Http.Headers;

IHeaderDictionary headersDictionary = httpContext.Request.Headers;

// GetTypedHeaders extension method provides strongly typed access to many headers
var requestHeaders = httpContext.Request.GetTypedHeaders();
CacheControlHeaderValue cacheControlHeaderValue = requestHeaders.CacheControl;

// For unknown header, unknownheaderValues has zero items and unknownheaderValue is ""
IList<string> unknownheaderValues = headersDictionary["unknownheader"];
string unknownheaderValue = headersDictionary["unknownheader"].ToString();

// For known header, knownheaderValues has 1 item and knownheaderValue is the value
IList<string> knownheaderValues = headersDictionary[HeaderNames.AcceptLanguage];
string knownheaderValue = headersDictionary[HeaderNames.AcceptLanguage].ToString();
```

HttpContext.Request.UserAgent translates to:

```
string userAgent = headersDictionary[HeaderNames.UserAgent].ToString();
```

HttpContext.Request.UrlReferrer translates to:

```
string urlReferrer = headersDictionary[HeaderNames.Referer].ToString();
```

HttpContext.Request.ContentType translates to:

```
// using Microsoft.Net.Http.Headers;

MediaTypeHeaderValue mediaHeaderValue = requestHeaders.ContentType;
string contentType = mediaHeaderValue?.MediaType;   // ex. application/x-www-form-urlencoded
string contentMainType = mediaHeaderValue?.Type;    // ex. application
string contentSubType = mediaHeaderValue?.SubType;  // ex. x-www-form-urlencoded

System.Text.Encoding requestEncoding = mediaHeaderValue?.Encoding;
```

HttpContext.Request.Form translates to:

```
if (httpContext.Request.HasFormContentType)
{
    IFormCollection form;
```

```
        form = httpContext.Request.Form; // sync
        // Or
        form = await httpContext.Request.ReadFormAsync(); // async

        string firstName = form["firstname"];
        string lastName = form["lastname"];
    }
```

> **:** Read form values only if the content sub type is *x-www-form-urlencoded* or *form-data*.

HttpContext.Request.InputStream translates to:

```
    string inputBody;
    using (var reader = new System.IO.StreamReader(
        httpContext.Request.Body, System.Text.Encoding.UTF8))
    {
        inputBody = reader.ReadToEnd();
    }
```

> **:** Use this code only in a handler type middleware, at the end of a pipeline.
> You can read the raw body as shown above only once per request. Middleware trying to read the body after the first read will read an empty body.
> This does not apply to reading a form as shown earlier, because that is done from a buffer.

HttpContext.Request.RequestContext.RouteData

RouteData is not available in middleware in RC1.

HttpContext.Response

HttpContext.Response.Status and HttpContext.Response.StatusDescription translate to:

```
    // using Microsoft.AspNet.Http;
    httpContext.Response.StatusCode = StatusCodes.Status200OK;
```

HttpContext.Response.ContentEncoding and HttpContext.Response.ContentType translate to:

```
    // using Microsoft.Net.Http.Headers;
    var mediaType = new MediaTypeHeaderValue("application/json");
    mediaType.Encoding = System.Text.Encoding.UTF8;
    httpContext.Response.ContentType = mediaType.ToString();
```

HttpContext.Response.ContentType on its own also translates to:

```
    httpContext.Response.ContentType = "text/html";
```

HttpContext.Response.Output translates to:

```
    string responseContent = GetResponseContent();
    await httpContext.Response.WriteAsync(responseContent);
```

HttpContext.Response.TransmitFile

Serving up a file is discussed here.

HttpContext.Response.Headers

Sending response headers is complicated by the fact that if you set them after anything has been written to the response body, they will not be sent.

The solution is to set a callback method that will be called right before writing to the response starts. This is best done at the start of the `Invoke` method in your middleware. It is this callback method that sets your response headers.

The following code sets a callback method called `SetHeaders`:

```csharp
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

The `SetHeaders` callback method would look like this:

```csharp
// using Microsoft.AspNet.Http.Headers;
// using Microsoft.Net.Http.Headers;

private Task SetHeaders(object context)
{
    var httpContext = (HttpContext)context;

    // Set header with single value
    httpContext.Response.Headers["ResponseHeaderName"] = "headerValue";

    // Set header with multiple values
    string[] responseHeaderValues = new string[] { "headerValue1", "headerValue1" };
    httpContext.Response.Headers["ResponseHeaderName"] = responseHeaderValues;

    // Translating ASP.NET 4's HttpContext.Response.RedirectLocation
    httpContext.Response.Headers[HeaderNames.Location] = "http://www.example.com";
    // Or
    httpContext.Response.Redirect("http://www.example.com");

    // GetTypedHeaders extension method provides strongly typed access to many headers
    var responseHeaders = httpContext.Response.GetTypedHeaders();

    // Translating ASP.NET 4's HttpContext.Response.CacheControl
    responseHeaders.CacheControl = new CacheControlHeaderValue
    {
        MaxAge = new System.TimeSpan(365, 0, 0, 0)
        // Many more properties available
    };

    // If you use .Net 4.6+, Task.CompletedTask will be a bit faster
    return Task.FromResult(0);
}
```

HttpContext.Response.Cookies

Cookies travel to the browser in a *Set-Cookie* response header. As a result, sending cookies requires the same callback as used for sending response headers:

```
    public async Task Invoke(HttpContext httpContext)
    {
        // ...
        httpContext.Response.OnStarting(SetCookies, state: httpContext);
        httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

The `SetCookies` callback method would look like the following:

```
    private Task SetCookies(object context)
    {
        var httpContext = (HttpContext)context;

        IResponseCookies responseCookies = httpContext.Response.Cookies;

        responseCookies.Append("cookie1name", "cookie1value");
        responseCookies.Append("cookie2name", "cookie2value",
            new CookieOptions { Expires = System.DateTime.Now.AddDays(5), HttpOnly = true });

        // If you use .Net 4.6+, Task.CompletedTask will be a bit faster
        return Task.FromResult(0);
    }
```

Additional Resources

- HTTP Handlers and HTTP Modules Overview
- *Configuration*
- *Application Startup*
- *Middleware*

1.14.6 Migrating from ASP.NET 5 RC1 to ASP.NET Core 1.0

By Cesar Blum Silveira, Rachel Appel, Rick Anderson

Sections:

- *Update Target Framework Monikers (TFMs)*
- *Namespace and package ID changes*
- *Commands and tools*
- *Hosting*
- *Kestrel*
- *ASP.NET 5 MVC compile views*
- *Configuration*
- *Logging*
- *Identity*
- *Working with IIS*
- *Updating Launch Settings in Visual Studio*
- *Server garbage collection (GC)*

ASP.NET 5 RC1 apps were based on the .NET Execution Environment (DNX) and made use of DNX specific features. ASP.NET Core 1.0 is based on .NET Core, so you must first migrate your application to the new .NET Core project model. See migrating from DNX to .NET Core CLI for more information.

See the following resources for a list of some of the most significant changes, announcements and migrations information:

- ASP.NET Core RC2 significant changes

- ASP.NET Core 1.0 significant changes

- Upgrading from Entity Framework RC1 to RTM

- *Migrating from ASP.NET Core RC2 to ASP.NET Core 1.0*

Update Target Framework Monikers (TFMs)

If your app targeted `dnx451` or `dnxcore50` in the `frameworks` section of *project.json*, you must make the following changes:

DNX	.NET Core
`dnx451`	`net451`
`dnxcore50`	`netcoreapp1.0`

.NET Core apps must add a dependency to the `Microsoft.NETCore.App` package:

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

Namespace and package ID changes

- ASP.NET 5 has been renamed to ASP.NET Core 1.0

- ASP.NET MVC and Identity are now part of ASP.NET Core

- ASP.NET MVC 6 is now ASP.NET Core MVC

- ASP.NET Identity 3 is now ASP.NET Core Identity

- ASP.NET Core 1.0 package versions are `1.0.0`

- ASP.NET Core 1.0 tool package versions are `1.0.0-preview2-final`

Namespace and package name changes:

ASP.NET 5 RC1	ASP.NET Core 1.0
`Microsoft.AspNet.*`	`Microsoft.AspNetCore.*`
`EntityFramework.*`	`Microsoft.EntityFrameworkCore.*`
`Microsoft.Data.Entity.*`	`Microsoft.EntityFrameworkCore.*`

The `EntityFramework.Commands` package is no longer available. The `ef` command is now available as a tool in the `Microsoft.EntityFrameworkCore.Tools` package.

The following packages have been renamed:

ASP.NET 5 RC1	ASP.NET Core 1.0
EntityFramework.MicrosoftSqlServer	Microsoft.EntityFrameworkCore.SqlServer
Microsoft.AspNet.Diagnostics.Entity	Microsoft.AspNetCore.Dianostics.EntityFrameworkCore
Microsoft.AspNet.Identity.EntityFramework	Microsoft.AspNetCore.Identity.EntityFrameworkCore
Microsoft.AspNet.Tooling.Razor	Microsoft.AspNetCore.Razor.Tools

Commands and tools

The `commands` section of the *project.json* file is no longer supported. Use `dotnet run` or `dotnet <DLL name>` instead.

.NET Core CLI has introduced the concept of tools. *project.json* now supports a `tools` section where packages containing tools can be specified. Some important functionality for ASP.NET Core 1.0 applications has been moved to tools.

See .NET Core CLI extensibility model for more information on .NET Core CLI tools.

Publishing to IIS

IIS publishing is now provided by the `publish-iis` tool in the `Microsoft.AspNetCore.Server.IISIntegration.Tools` package. If you intend to run your app behind IIS, add the `publish-iis` tool to your *project.json*:

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

The `publish-iis` tool is commonly used in the `postpublish` script in *project.json*:

```
{
  "postpublish": [ "dotnet publish-iis --publish-folder %publish:OutputPath% --framework %publish:Ful
}
```

Entity Framework commands

The `ef` tool is now provided in the `Microsoft.EntityFrameworkCore.Tools` package:

```
{
  "tools": {
    "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final"
  }
}
```

For more information, see .NET Core CLI.

Razor tools

Razor tooling is now provided in the `Microsoft.AspNetCore.Razor.Tools` package:

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

SQL cache tool

The `sqlservercache` command, formerly provided by the `Microsoft.Extensions.Caching.SqlConfig` package, has been replaced by the `sql-cache` tool, available through the `Microsoft.Extensions.Caching.SqlConfig.Tools` package:

```
{
  "tools": {
    "Microsoft.Extensions.Caching.SqlConfig.Tools": "1.0.0-preview2-final"
  }
}
```

User secrets manager

The `user-secret` command, formerly provided by the `Microsoft.Extensions.SecretManager` package, has been replaced by the `user-secrets` tool, available through the `Microsoft.Extensions.SecretManager.Tools` package:

```
{
  "tools": {
    "Microsoft.Extensions.SecretManager.Tools": "1.0.0-preview2-final"
  }
}
```

File watcher

The `watch` command, formerly provided by the `Microsoft.Dnx.Watcher` package, has been replaced by the `watch` tool, available through the `Microsoft.DotNet.Watcher.Tools` package:

```
{
  "tools": {
    "Microsoft.DotNet.Watcher.Tools": "1.0.0-preview2-final"
  }
}
```

For more information on the file watcher, see **Dotnet watch** in *Tutorials*.

Hosting

Creating the web application host

ASP.NET Core 1.0 apps are console apps; you must define an entry point for your app that sets up a web host and runs it. Below is an example from the startup code for one of the Web Application templates in Visual Studio:

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseContentRoot(Directory.GetCurrentDirectory())
```

```
            .UseIISIntegration()
            .UseStartup<Startup>()
            .Build();

        host.Run();
    }
}
```

You must add the `emitEntryPoint` to the `buildOptions` section of your application's *project.json*:

```
{
  "buildOptions": {
    "emitEntryPoint": true
  }
}
```

Class and interface renames

All classes and interfaces prefixed with `WebApplication` have been renamed to start with `WebHost`:

ASP.NET 5 RC1	ASP.NET Core 1.0
IWebApplicationBuilder	IWebHostBuilder
WebApplicationBuilder	WebHostBuilder
IWebApplication	IWebHost
WebApplication	WebHost
WebApplicationOptions	WebHostOptions
WebApplicationDefaults	WebHostDefaults
WebApplicationService	WebHostService
WebApplicationConfiguration	WebHostConfiguration

Content root and web root

The application base path is now called the content root.

The web root of your application is no longer specified in your *project.json* file. It is defined when setting up the web host and defaults to `wwwroot`. Call the `UseWebRoot` extension method to specify a different web root folder. Alternatively, you can specify the web root folder in configuration and call the `UseConfiguration` extension method.

Server address binding

The server addresses that your application listens on can be specified using the `UseUrls` extension method or through configuration.

Specifying only a port number as a binding address is no longer supported. The default binding address is http://localhost:5000

Hosting configuration

The `UseDefaultHostingConfiguration` method is no longer available. The only configuration values read by default by `WebHostBuilder` are those specified in environment variables prefixed with `ASPNETCORE_*`. All other

configuration sources must now be added explicitly to an `IConfigurationBuilder` instance. See *Configuration* for more information.

The environment key is set with the `ASPNETCORE_ENVIRONMENT` environment variable. `ASPNET_ENV` and `Hosting:Environment` are still supported, but generate a deprecated message warning.

Hosting service changes

Dependency injection code that uses `IApplicationEnvironment` must now use `IHostingEnvironment`. For example, in your `Startup` class, change:

```
public Startup(IApplicationEnvironment applicationEnvironment)
```

To:

```
public Startup(IHostingEnvironment hostingEnvironment)
```

Kestrel

Kestrel configuration has changed. This GitHub announcement outlines the changes you must make to configure Kestrel if you are not using default settings.

Controller and action results renamed

The following `Controller` methods have been renamed and moved to `ControllerBase`:

ASP.NET 5 RC1	ASP.NET Core 1.0
HttpUnauthorized	Unauthorized
HttpNotFound (and its overloads)	NotFound
HttpBadRequest (and its overloads)	BadRequest

The following action result types have also been renamed:

ASP.NET 5 RC1	ASP.NET Core 1.0
Microsoft.AspNet.Mvc.HttpOkObjectResult	Microsoft.AspNetCore.Mvc.OkObjectResult
Microsoft.AspNet.Mvc.HttpOkResult	Microsoft.AspNetCore.Mvc.OkResult
Microsoft.AspNet.Mvc.HttpNotFoundObjectResult	Microsoft.AspNetCore.Mvc.NotFoundObjectResult
Microsoft.AspNet.Mvc.HttpNotFoundResult	Microsoft.AspNetCore.Mvc.NotFoundResult
Microsoft.AspNet.Mvc.HttpStatusCodeResult	Microsoft.AspNetCore.Mvc.StatusCodeResult
Microsoft.AspNet.Mvc.HttpUnauthorizedResult	Microsoft.AspNetCore.Mvc.UnauthorizedResult

ASP.NET 5 MVC compile views

To compile views, set the `preserveCompilationContext` option in *project.json* to preserve the compilation context, as shown here:

```
{
  "buildOptions": {
    "preserveCompilationContext": true
  }
}
```

Changes in views

Views now support relative paths.

The Validation Summary Tag Helper `asp-validation-summary` attribute value has changed. Change:

```
<div asp-validation-summary="ValidationSummary.All"></div>
```

To:

```
<div asp-validation-summary="All"></div>
```

Changes in ViewComponents

- The sync APIs have been removed

- `Component.Render()`, `Component.RenderAsync()`, and `Component.Invoke()` have been removed

- To reduce ambiguity in View Component method selection, we've modified the selection to only allow exactly one `Invoke()` or `InvokeAsync()` per View Component

- `InvokeAsync()` now takes an anonymous object instead of separate parameters

- To use a view component, call `@Component.InvokeAsync("Name of view component", <parameters>)` from a view. The parameters will be passed to the `InvokeAsync()` method. The following example demonstrates the `InvokeAsync()` method call with two parameters:

ASP.NET 5 RC1:

```
@Component.InvokeAsync("Test", "MyName", 15)
```

ASP.NET Core 1.0:

```
@Component.InvokeAsync("Test", new { name = "MyName", age = 15 })
@Component.InvokeAsync("Test", new Dictionary<string, object> {
                        ["name"] = "MyName", ["age"] = 15 })
@Component.InvokeAsync<TestViewComponent>(new { name = "MyName", age = 15})
```

Updated controller discovery rules

There are changes that simplify controller discovery:

The new `ControllerAttribute` can be used to mark a class (and it's subclasses) as a controller. A class whose name doesn't end in `Controller` and derives from a base class that ends in `Controller` is no longer considered a controller. In this scenario, `ControllerAttribute` must be applied to the derived class itself or to the base class.

A type is considered a controller if **all** the following conditions are met:

- The type is a public, concrete, non-open generic class

- `NonControllerAttribute` is **not** applied to any type in its hierarchy

- The type name ends with `Controller`, or `ControllerAttribute` is applied to the type or one of its ancestors.

: If `NonControllerAttribute` is applied anywhere in the type hierarchy, the discovery conventions will never consider that type or its descendants to be a controller. In other words, `NonControllerAttribute` takes precedence over `ControllerAttribute`.

Configuration

The `IConfigurationSource` interface has been introduced to represent the configuration used to build an `IConfigurationProvider`. It is no longer possible to access the provider instances from `IConfigurationBuilder`, only the sources. This is intentional, and may cause loss of functionality as you can no longer do things like call `Load` on the provider instances.

File-based configuration providers support both relative and absolute paths to configuration files. If you want to specify file paths relative to your application's content root, you must call the `SetBasePath` extension method on `IConfigurationBuilder`:

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json");
}
```

Automatic reload on change

The `IConfigurationRoot.ReloadOnChanged` extension method is no longer available. File-based configuration providers now provide extension methods to `IConfigurationBuilder` that allow you to specify whether configuration from those providers should be reloaded when there are changes in their files. See `AddJsonFile`, `AddXmlFile` and `AddIniFile` for details.

Logging

`LogLevel.Verbose` has been renamed to `Trace` and is now considered less severe than `Debug`.

The `MinimumLevel` property has been removed from `ILoggerFactory`. Each logging provider now provides extension methods to `ILoggerFactory` that allow specifying a minimum logging level. See `AddConsole`, `AddDebug`, and `AddEventLog` for details.

Identity

The signatures for the following methods or properties have changed:

ASP.NET 5 RC1	ASP.NET Core 1.0
ExternalLoginInfo.ExternalPrincipal	ExternalLoginInfo.Principal
User.IsSignedIn()	SignInManager.IsSignedIn(User)
UserManager.FindByIdAsync(HttpContext.User.GetUserId())	UserManager.GetUserAsync(HttpContext.User)
User.GetUserId()	UserManager.GetUserId(User)

To use Identity in a view, add the following:

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<TUser> SignInManager
@inject UserManager<TUser> UserManager
```

Working with IIS

The package `Microsoft.AspNetCore.IISPlatformHandler` has been replaced by `Microsoft.AspNetCore.Server.IISIntegration`.

HttpPlatformHandler has been replaced by the *ASP.NET Core Module (ANCM)*. The *web.config* file created by the *Publish to IIS tool* now configures IIS to the ANCM instead of HttpPlatformHandler to reverse-proxy requests.

The ASP.NET Core Module must be configured in *web.config*:

```xml
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified"/
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%" arguments="%LAUNCHER_ARGS%"
                stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout"
                forwardWindowsAuthToken="false"/>
  </system.webServer>
</configuration>
```

The *Publish to IIS tool* generates a correct *web.config*. See *Publishing to IIS* for more details.

IIS integration middleware is now configured when creating the `Microsoft.AspNetCore.Hosting.WebHostBuilder`, and is no longer called in the `Configure` method of the `Startup` class:

```csharp
var host = new WebHostBuilder()
    .UseIISIntegration()
    .Build();
```

Web Deploy changes

Delete any *<app name> - Web Deploy-publish.ps1* scripts created with Visual Studio web deploy using ASP.NET 5 RC1. The ASP.NET 5 RC1 scripts (which are DNX based) are not compatible with dotnet based scripts. Use Visual Studio to generate new web deploy scripts.

applicationhost.config changes

An *applicationhost.config* file created with ASP.NET 5 RC1 will point ASP.NET Core to an invalid *content root* location. With such a *applicationhost.config* file, ASP.NET Core will be configured with *content root*/*web root* as the *content root* folder and therefore look for *web.config* in `Content root/wwwroot`. The *web.config* file must be in the *content root* folder. When configured like this, the app will terminate with an HTTP 500 error.

Updating Launch Settings in Visual Studio

Update `launchSettings.json` to remove the web target and add the following:

```
{
  "WebApplication1": {
    "commandName": "Project",
    "launchBrowser": true,
    "launchUrl": "http://localhost:5000",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  }
}
```

Server garbage collection (GC)

You must turn on server garbage collection in *project.json* or *app.config* when running ASP.NET projects on the full .NET Framework:

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.Server": true
    }
  }
}
```

1.14.7 Migrating from ASP.NET Core RC2 to ASP.NET Core 1.0

By Cesar Blum Silveira

Sections:

- *Overview*
- *Tools*
- *Hosting*
- *ASP.NET MVC Core*
- *Security*

Overview

This migration guide covers migrating an ASP.NET Core RC2 application to ASP.NET Core 1.0.

There weren't many significant changes to ASP.NET Core between the RC2 and 1.0 releases. For a complete list of changes, see the ASP.NET Core 1.0 announcements.

Tools

For the tools we ship, you no longer need to use imports in *project.json*. For example:

```
{
  "tools": {
    "Microsoft.AspNetCore.Server.IISIntegration.Tools": {
      "version": "1.0.0-preview1-final",
```

```
        "imports": "portable-net45+win8+dnxcore50"
    }
  }
}
```

Becomes:

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

Hosting

The `UseServer` is no longer available for `IWebHostBuilder`. You must now use `UseKestrel` or `UseWebListener`.

ASP.NET MVC Core

The `HtmlEncodedString` class has been replaced by `HtmlString` (contained in the `Microsoft.AspNetCore.Html.Abstractions` package).

Security

The `AuthorizationHandler<TRequirement>` class now only contains an asynchronous interface.

1.15 Contribute

1.15.1 ASP.NET Docs Style Guide

By Steve Smith

This document provides an overview of how articles published on docs.asp.net should be formatted. You can actually use this file, itself, as a template when contributing articles.

> **Sections:**
>
> - *Article Structure*
> - *ReStructuredText Syntax*
> - *Additional Reading*
> - *Summary*

Article Structure

Articles should be submitted as individual text files with a **.rst** extension. Authors should be sure they are familiar with the Sphinx Style Guide, but where there are disagreements, this document takes precedence. The article should begin with its title on line 1, followed by a line of === characters. Next, the author should be displayed with a link to an author specific page (ex. the author's GitHub user page, Twitter page, etc.).

Articles should typically begin with a brief abstract describing what will be covered, followed by a bulleted list of topics, if appropriate. If the article has associated sample files, a link to the samples should be included following this bulleted list.

Articles should typically include a Summary section at the end, and optionally additional sections like Next Steps or Additional Resources. These should not be included in the bulleted list of topics, however.

Headings

Typically articles will use at most 3 levels of headings. The title of the document is the highest level heading and must appear on lines 1-2 of the document. The title is designated by a row of === characters.

Section headings should correspond to the bulleted list of topics set out after the article abstract. *Article Structure*, above, is an example of a section heading. A section heading should appear on its own line, followed by a line consisting of — characters.

Subsection headings can be used to organize content within a section. *Headings*, above, is an example of a subsection heading. A subsection heading should appear on its own line, followed by a line of ^^^ characters.

```
Title (H1)
==========

Section heading (H2)
--------------------

Subsection heading (H3)
^^^^^^^^^^^^^^^^^^^^^^^
```

For section headings, only the first word should be capitalized:

- Use this heading style
- Do Not Use This Style

More on sections and headings in ReStructuredText: http://sphinx-doc.org/rest.html#sections

ReStructuredText Syntax

The following ReStructuredText elements are commonly used in ASP.NET documentation articles. Note that **indentation and blank lines are significant!**

Inline Markup

Surround text with:

- One asterisk for *emphasis* (*italics*)
- Two asterisks for **strong emphasis** (**bold**)
- Two backticks for ``code samples``(an `<html>` element)

: Inline markup cannot be nested, nor can surrounded content start or end with whitespace (`* foo*` is wrong).

Escaping is done using the \ backslash.

Format specific items using these rules:

- *Italics* (surround with *) - Files, folders, paths (for long items, split onto their own line) - New terms - URLs (unless rendered as links, which is the default)

- **Strong** (surround with **) - UI elements

- `Code Elements` (surround with '') - Classes and members - Command-line commands - Database table and column names - Language keywords

Links

Links should use HTTPS when possible. Inline hyperlinks are formatted like this:

```
Learn more about `ASP.NET <https://www.asp.net>`_.
```

Learn more about [ASP.NET](https://www.asp.net).

Surround the link text with backticks. Within the backticks, place the target in angle brackets, and ensure there is a space between the end of the link text and the opening angle bracket. Follow the closing backtick with an underscore.

In addition to URLs, documents and document sections can also be linked by name:

```
For example, here is a link to the `Inline Markup`_ section, above.
```

For example, here is a link to the *Inline Markup* section, above.

Any element that is rendered as a link should not have any additional formatting or styling.

Lists

Lists can be started with a – or * character:

```
- This is one item
- This is a second item
```

Numbered lists can start with a number, or they can be auto numbered by starting each item with the # character. Please use the # syntax.

```
1. Numbered list item one.(don't use numbers)
2. Numbered list item two.(don't use numbers)

#. Auto-numbered one.
#. Auto-numbered two.
```

Source Code

Source code is very commonly included in these articles. Images should never be used to display source code. Prefer `literalinclude` for most code samples. Reserve `code-block` for small snippets that are not included in the sample project. A `code-block` can be declared as shown below, including spaces, blank lines, and indentation:

```
.. code-block:: c#

public void Foo()
{
  // Foo all the things!
}
```

This results in:

```
public void Foo()
{
    // Foo all the things!
}
```

The code block ends when you begin a new paragraph without indentation. Sphinx supports quite a few different languages. Some common language strings that are available include:

- c#

- javascript

- html

Line numbers should only be used while editing to assist in find the line numbers to emphasize. Code blocks also support line numbers and emphasizing or highlighting certain lines:

```
.. code-block:: c#
  :linenos:
  :emphasize-lines: 3

  public void Foo()
  {
    // Foo all the things!
  }
```

This results in:

```
1   public void Foo()
2   {
3       // Foo all the things!
4   }
```

: Once the `emphasize-lines` is determined, remove `:linenos:`. When updating a doc, remove all occurrences of `:linenos:`.

: `caption` and `name` will result in a code-block not being displayed due to our builds using a Sphinx version prior to version 1.3. If you don't see a code block displayed above this note, it's most likely because the version of Sphinx is < 1.3.

Images

Images such as screen shots and explanatory figures or diagrams should be placed in a `_static` folder within a folder named the same as the article file. References to images should therefore always be made using relative references, e.g. `article-name/style-guide/_static/asp-net.png`. Note that images should always be saved as all lower-case file names, using hyphens to separate words, if necessary.

: Do not use images for code. Use `code-block` or `literalinclude` instead.

To include an image in an article, use the `..` `image` directive:

```
.. image:: style-guide/_static/asp-net.png
```

: No quotes are needed around the file name.

Here's an example using the above syntax:

Images are responsively sized according to the browser viewport when using this directive. Currently the maximum width supported by the https://docs.asp.net theme is 697px.

Notes

To add a note callout, like the ones shown in this document, use the `..   note::` directive.

```
.. note:: This is a note.
```

This results in:

: This is a note.

Including External Source Files

One nice feature of ReStructuredText is its ability to reference external files. This allows actual sample source files to be referenced from documentation articles, reducing the chances of the documentation content getting out of sync with the actual, working code sample (assuming the code sample works, of course). However, if documentation articles are referencing samples by filename and line number, it is important that the documentation articles be reviewed whenever changes are made to the source code, otherwise these references may be broken or point to the wrong line number. For this reason, it is recommended that samples be specific to individual articles, so that updates to the sample will only affect a single article (at most, an article series could reference a common sample). Samples should therefore be placed in a subfolder named the same as the article file, in a `sample` folder (e.g. `/article-name/sample/`).

External file references can specify a language, emphasize certain lines, display line numbers (recommended), similar to *Source Code*. Remember that these line number references may need to be updated if the source file is changed.

```
.. literalinclude:: style-guide/_static/startup.cs
  :language: c#
  :emphasize-lines: 19,25-27
  :linenos:
```

```csharp
1   using System;
2   using Microsoft.AspNet.Builder;
3   using Microsoft.AspNet.Hosting;
4   using Microsoft.AspNet.Http;
5   using Microsoft.Framework.DependencyInjection;
6
7   namespace ProductsDnx
8   {
9       public class Startup
10      {
11          public Startup(IHostingEnvironment env)
12          {
13          }
14
15          // This method gets called by a runtime.
16          // Use this method to add services to the container
17          public void ConfigureServices(IServiceCollection services)
18          {
19              services.AddMvc();
20          }
21
22          // Configure is called after ConfigureServices is called.
23          public void Configure(IApplicationBuilder app, IHostingEnvironment env)
24          {
25              app.UseStaticFiles();
26              // Add MVC to the request pipeline.
27              app.UseMvc();
28          }
29      }
30  }
```

You can also include just a section of a larger file, if desired:

```
.. literalinclude:: style-guide/_static/startup.cs
  :language: c#
  :lines: 1,4,20-
  :linenos:
```

This would include the first and fourth line, and then line 20 through the end of the file.

Literal includes also support *Captions* and names, as with `code-block` elements. If the `caption` is left blank, the file name will be used as the caption. Note that captions and names are available with Sphinx 1.3, which the ReadTheDocs theme used by this system is not yet updated to support.

Format code to eliminate or minimize horizontal scroll bars.

Tables

Tables should never render with horizontal scroll bars. Tables can be constructed using grid-like "ASCII Art" style text. In general they should only be used where it makes sense to present some tabular data. Rather than include all of the

syntax options here, you will find a detailed reference at http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html#grid-tables.

UI navigation

When documenting how a user should navigate a series of menus, use the `:menuselection:` directive:

```
:menuselection:`Windows --> Views --> Other...`
```

This will result in *Windows → Views → Other...*.

Additional Reading

Learn more about Sphinx and ReStructuredText:

- Sphinx documentation
- RST Quick Reference

Summary

This style guide is intended to help contributors quickly create new articles for docs.asp.net. It includes the most common RST syntax elements that are used, as well as overall document organization guidance. If you discover mistakes or gaps in this guide, please submit an issue.

Related Resources

- .NET Core
- Entity Framework Core
- WebHooks

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*

R

RFC
   RFC 2616#section-14.42, 234