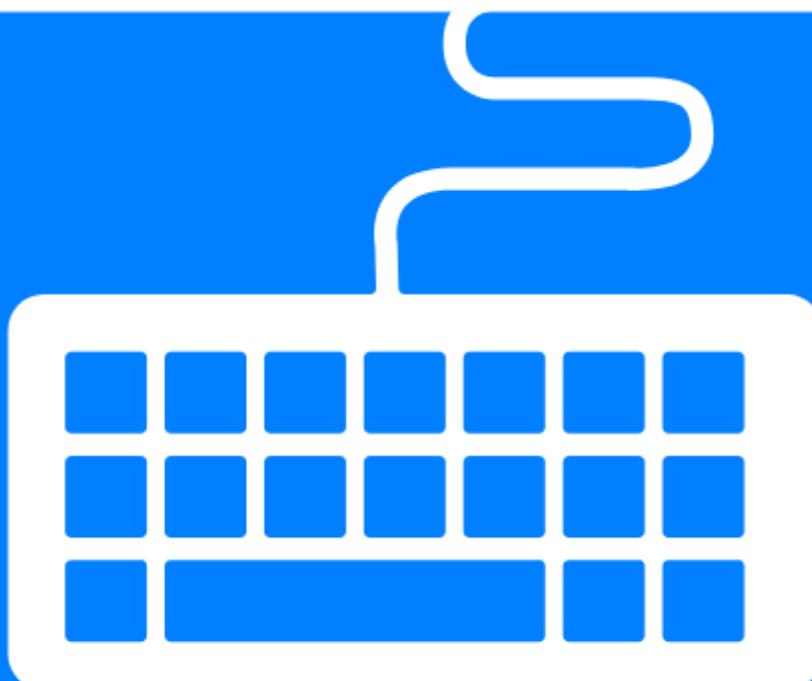


As seen on  
[WakeUpAndCode.com](https://WakeUpAndCode.com)

26 topics for ASP.NET Core 3.0  
Web App Development

## **ASP.NET Core A-Z**

**Shahed Chowdhuri**



# Authentication & Authorization in ASP .NET Core

By Shahed C on January 7, 2019

8 Replies

This is the first of a new series of posts on ASP .NET Core for 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



**A – Z of ASP .NET Core!**

## In this Article:

- A is for Authentication & Authorization
- Authentication in ASP .NET Core
- Authentication in NetLearner
- Authorization in ASP.NET Core (MVC)
- Authorization in ASP.NET Core (Razor Pages)
- Testing Authorization in NetLearner
- Other Authorization Options
- References

# A is for Authentication & Authorization

*Authentication* and *Authorization* are two different things, but they also go hand in hand. Think of Authentication as letting someone into your home and Authorization as allowing your guests to do specific things once they're inside (e.g. wear their shoes indoors, eat your food, etc). In other words, Authentication lets your web app's users identify themselves to get access to your app and Authorization allows them to get access to specific features and functionality.

In this article, we will take a look at the NetLearner app, on how specific pages can be restricted to users who are logged in to the application. Throughout the series, I will try to focus on new code added to NetLearner or build a smaller sample app if necessary.

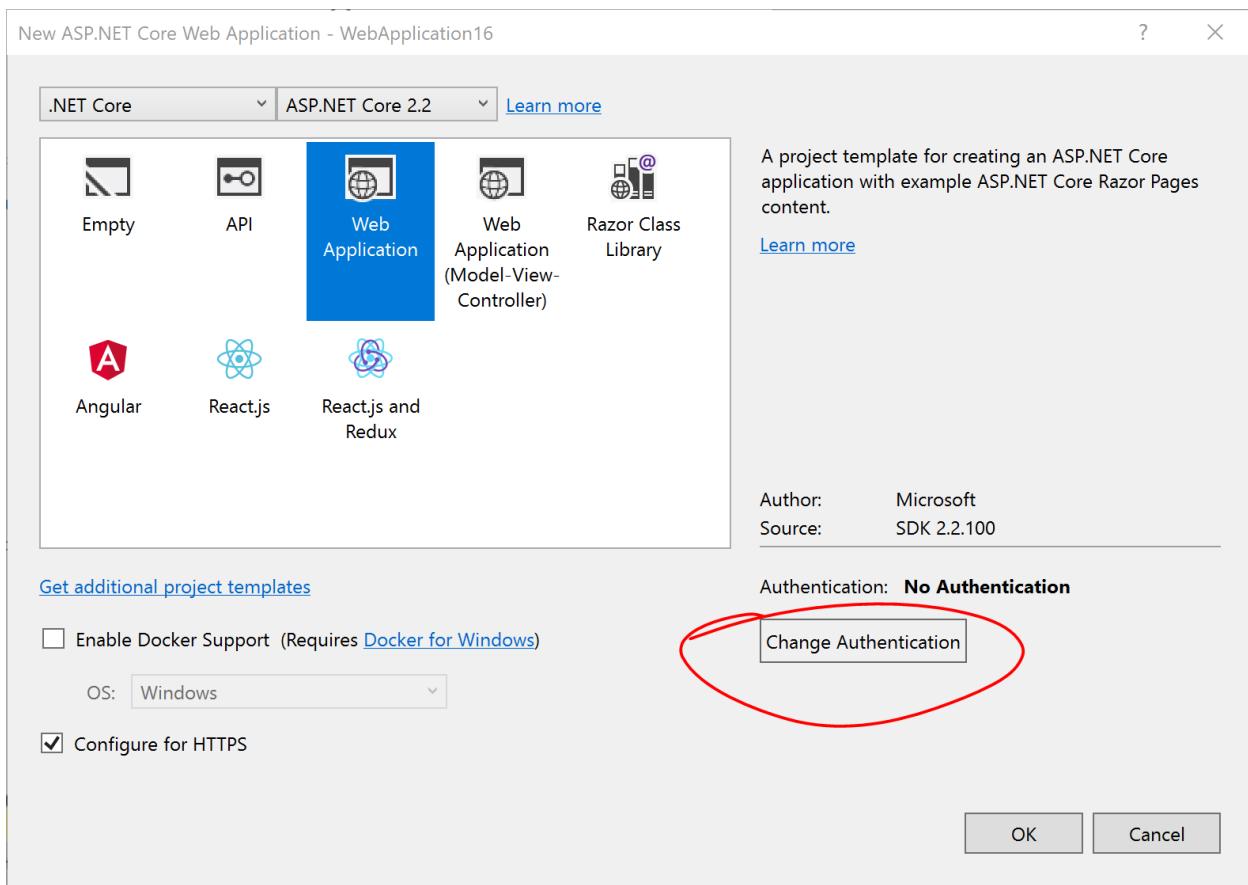
## Authentication in ASP .NET Core

The quickest way to add authentication to your ASP .NET Core app is to use of the pre-built templates with one of the Authentication options. The examples below demonstrate both the CLI commands and Visual Studio UI.

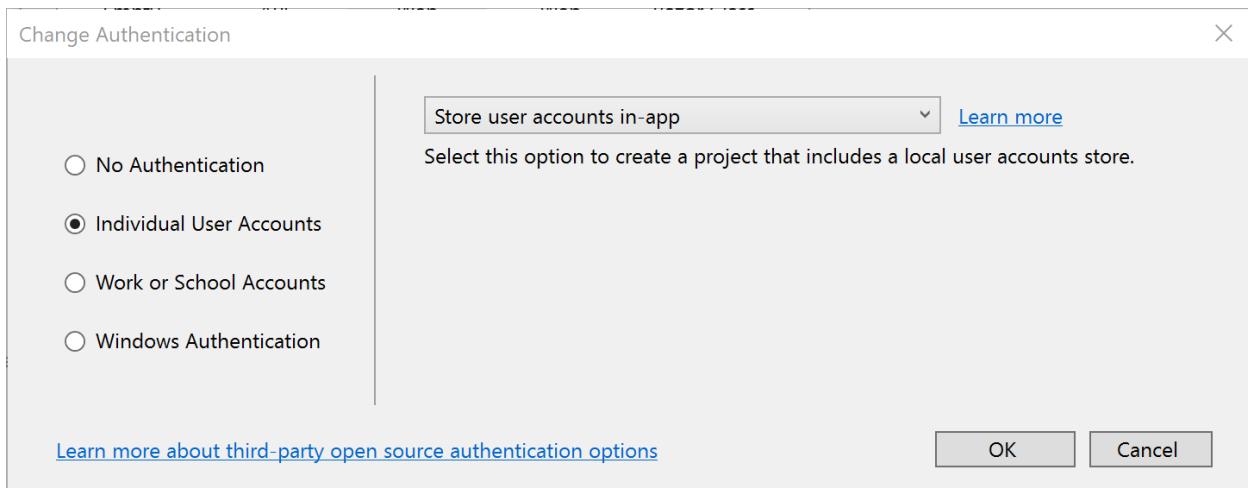
### *CLI Commands:*

```
> dotnet new webapp --auth Individual
```

### *Visual Studio 2017 new project with Authentication:*



Change Authentication upon creating a new project



Select Authentication Type

The above example uses “Individual” authentication, which offers a couple of options:

- **Store user accounts in-app:** includes a local user accounts store
- **Connect to an existing user store in the cloud:** connect to an existing Azure AD B2C application

Even if I choose to start with a local database, I can update the connection string to point to a SQL Server instance on my network or in the cloud, depending on which configuration is being loaded. If you're wondering where your Identity code lives, check out my previous post on Razor UI Libraries, and jump to the last section where Identity is mentioned.

From the documentation, the types of authentication are listed below.

- **None:** No authentication
- **Individual:** Individual authentication
- **IndividualB2C:** Individual authentication with Azure AD B2C
- **SingleOrg:** Organizational authentication for a single tenant
- **MultiOrg:** Organizational authentication for multiple tenants
- **Windows:** Windows authentication

To get help information about Authentication types, simply type —help after the —auth flag, e.g.

```
> dotnet new webapp --auth --help
```

## Authentication in NetLearner

Within my NetLearner app, the following snippets of code are added to the Startup.cs configuration:

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

```

        .AddDefaultUI (UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationContext>();

    ...
} public void Configure (IApplicationBuilder app, IHostingEnvironment
env)
{
    ...
    app.UseStaticFiles ();
    ...
    app.UseAuthentication ();
    ...
    app.UseMvc ();
}

```

In the above, note that:

- The **ConfigureServices()** method has calls to services.**AddDbContext** and **server.AddDefaultIdentity**. The call to add a DB Context will vary depending on which data store you choose for authentication. The call to AddDefaultIdentity ensures that your app calls **AddIdentity**, **AddDefaultUI** and **AddDefaultTokenProviders** to add common identity features and user Register/Login functionality.
- The **Configure()** method has a call to **app.UseAuthentication** to ensure that authentication is used by your web app. Note that this appears *after* **app.UseStaticFiles** but *before* **app.UseMvc** to ensure that static files (html, css, js, etc) can be served without any authentication but MVC application-controlled routes and views/pages will follow authentication rules.

## Authorization in ASP.NET Core (MVC)

Even after adding authentication to a web app using the project template options, we can still access many parts of the application without having to log in. In order to restrict specific parts of the application, we will implement Authorization in our app.

If you've already worked with ASP .NET Core MVC apps before, you may be familiar with the **[Authorize]** attribute. This attribute can be added to a controller at the class level or even to specific action methods within a class.

```

[Authorize]
public class SomeController1: Controller
{
    // this controller class requires authentication
    // for all action methods
    public ActionResult SomeMethod()
    {
        //
    }
}

public class SomeController2: Controller
{
    public ActionResult SomeOpenMethod()
    {

    }

    [Authorize]
    public ActionResult SomeSecureMethod()
    {
        // this action method requires authentication
    }
}

```

*Well, what about Razor Pages in ASP .NET Core? If there are no controller classes, where would you add the **[Authorize]** attribute?*

## Authorization in ASP.NET Core (Razor Pages)

For Razor Pages, the quickest way to add Authorization for your pages (or entire folders of pages) is to update your **ConfigureServices()** method in your Startup.cs class, by calling **AddRazorPagesOptions()** after **AddMvc()**. The NetLearner configuration includes the following code:

```

services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    options.Conventions.AuthorizePage("/LearningResources/Create");
    options.Conventions.AuthorizePage("/LearningResources/Edit");
    options.Conventions.AuthorizePage("/LearningResources/Delete");
    options.Conventions.AllowAnonymousToPage("/Index");
})
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

```

The above code ensures that the CRUD pages for Creating, Editing and Deleting any of the LearningResources are only accessible to someone who is currently logged in. Each call to **AuthorizePage()** includes a specific page name identified by a known route. In this case, the LearningResources folder exists within the Pages folder of the application.

Finally, the call to **AllowAnonymousPage()** ensures that the app's index page (at the root of the Pages folder) is accessible to any user without requiring any login.

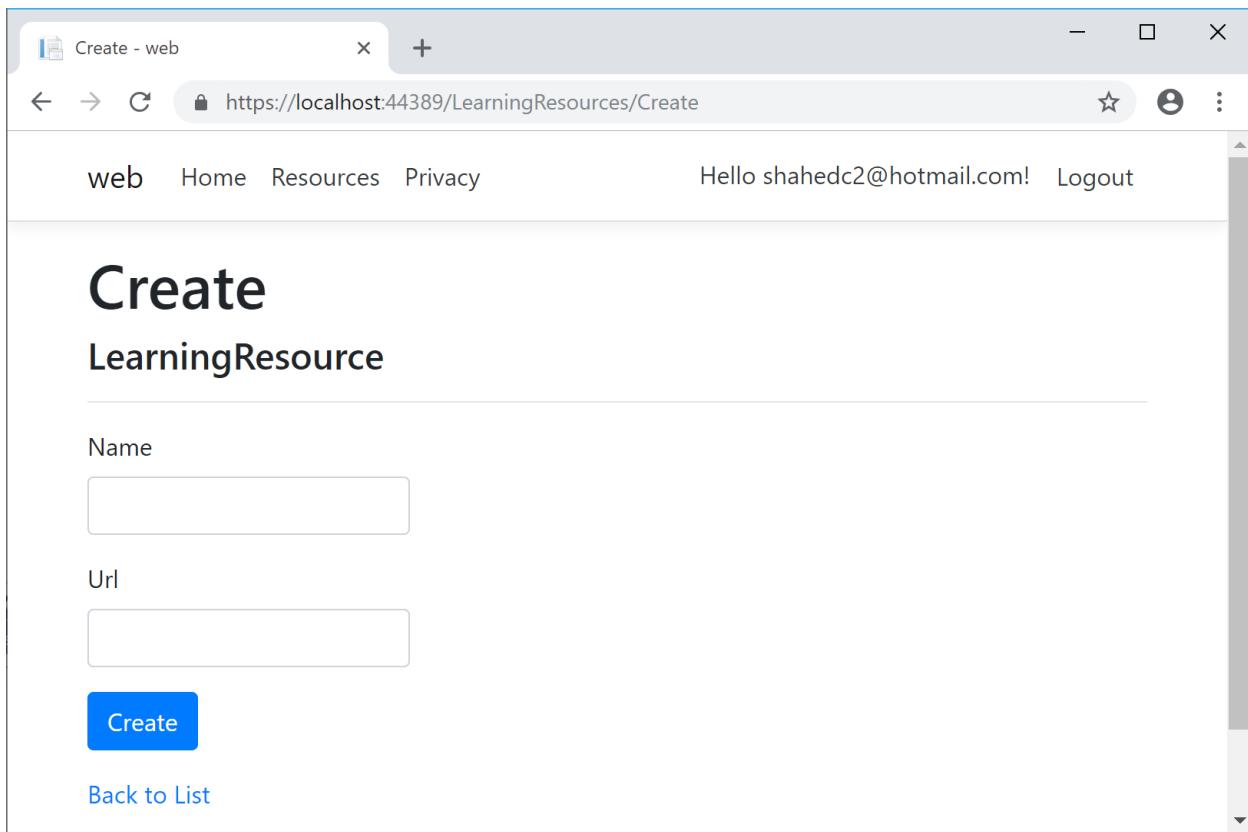
If you still wish to use the **[Authorize]** attribute for Razor Pages, you may apply this attribute in your PageModel classes for each Razor Page, as needed. If I were to add it to one of my Razor Pages in the LearningResources folder, it could look like this:

```
[Authorize]
public class CreateModel : PageModel
{
    ...
}
```

## Testing Authorization in NetLearner

When I run my application, I can register and log in as a user to create new Learning Resources. On first launch, I have to apply migrations to create the database from scratch. Please refer to my previous post on EF Core Migrations to learn how you can do the same in your environment.

Here's a screenshot of the Create page for a user who is logged in:



Here's a screenshot of the page that an “anonymous” user sees when no one is logged in, indicating that the user has been redirected to the Login page:

The screenshot shows a web browser window with the title "Log in - web". The URL in the address bar is <https://localhost:44389/Identity/Account/Login?ReturnUrl=%2FLearningResources%2F...>. The page content is a login form. At the top left, there are links for "Home", "Resources", and "Privacy". At the top right, there are links for "Register" and "Login". The main heading is "Log in". Below it, there are two options: "Use a local account to log in." and "Use another service to log in.". The "Use a local account" section contains fields for "Email" and "Password", and a "Remember me?" checkbox. A blue "Log in" button is located below these fields. The "Use another service to log in." section contains a message stating there are no external authentication services configured, with a link to "this article" for details.

Log in

Use a local account to log in.

Email

Password

Remember me?

Log in

Use another service to log in.

There are no external authentication services configured. See [this article](#) for details on setting up this ASP.NET application to support logging in via external services.

web Home Resources Privacy Register Login

Here's a screenshot of a list of Learning Resources, visible to anyone whether they're logged in or not:

The screenshot shows a web browser window titled "Index - web". The address bar displays the URL "https://localhost:44389/LearningResources". The page content includes a navigation bar with links for "web", "Home", "Resources", and "Privacy". On the right side of the navigation bar, there is a greeting "Hello shahedc2@hotmail.com!" and a "Logout" link. The main content area is titled "Index" and contains a "Create New" button. Below this is a table with three rows, each representing a resource entry. The columns are "Name" and "Url". The entries are: "Wake Up And Code!" with Url "https://wakeupandcode.com", "Microsoft Docs" with Url "https://docs.microsoft.com", and "Channel 9" with Url "https://channel9.msdn.com". Each row has "Edit | Details | Delete" links. At the bottom of the page, there is a copyright notice "© 2019 - web - Privacy".

Name	Url	
Wake Up And Code!	<a href="https://wakeupandcode.com">https://wakeupandcode.com</a>	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Microsoft Docs	<a href="https://docs.microsoft.com">https://docs.microsoft.com</a>	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Channel 9	<a href="https://channel9.msdn.com">https://channel9.msdn.com</a>	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

## Other Authorization Options

Razor Pages have multiple ways of restricting access to pages and folders, including the following methods (*as described in the official docs*):

- `AuthorizePage`: Require authorization to access a page
- `AuthorizeFolder`: Require authorization to access a folder of pages
- `AuthorizeAreaPage`: Require authorization to access an area page
- `AuthorizeAreaFolder`: Require authorization to access a folder of areas
- `AllowAnonymousToPage`: Allow anonymous access to a page
- `AllowAnonymousToFolder`: Allow anonymous access to a folder of pages

You can get more information on all of the above methods at the following URL:

- Razor Pages authorization conventions in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/razor-pages-authorization>
- Detailed Tutorial: <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/secure-data>

## References

To learn more about Authentication, Authorization and other related topics (e.g. Roles and Claims), check out the official docs:

- Razor Pages authorization conventions in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/razor-pages-authorization>
- Is the Authorize attribute needed in Razor Pages? What about Roles, Claims and Policies?: <https://github.com/aspnet/Docs/issues/6301>
- [Authorize] Filter methods for Razor Pages in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/razor-pages/filter/#authorize-filter-attribute>
- Simple authorization in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/simple>
- Role-based authorization in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles>
- Claims-based authorization in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/claims>
- Policy-based authorization in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies>

# Blazor Full-Stack Web Dev in ASP .NET Core

By Shahed C on January 14, 2019

11 Replies

This is the second of a new series of posts on ASP .NET Core for 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

- B is for Blazor
- Entry Point and Configuration
- Client-Side Application
- LifeCycle Methods
- Updating the UI
- Next Steps
- References

# B is for Blazor

In a previous post, I covered various types of Pages that you will see when developing ASP .NET Core web applications. In this article, we will take a look at a Blazor sample and see how it works. Blazor (Browser + Razor) is an experimental .NET web framework which allows you to write full-stack C# .NET web applications that run in a web browser, using WebAssembly.

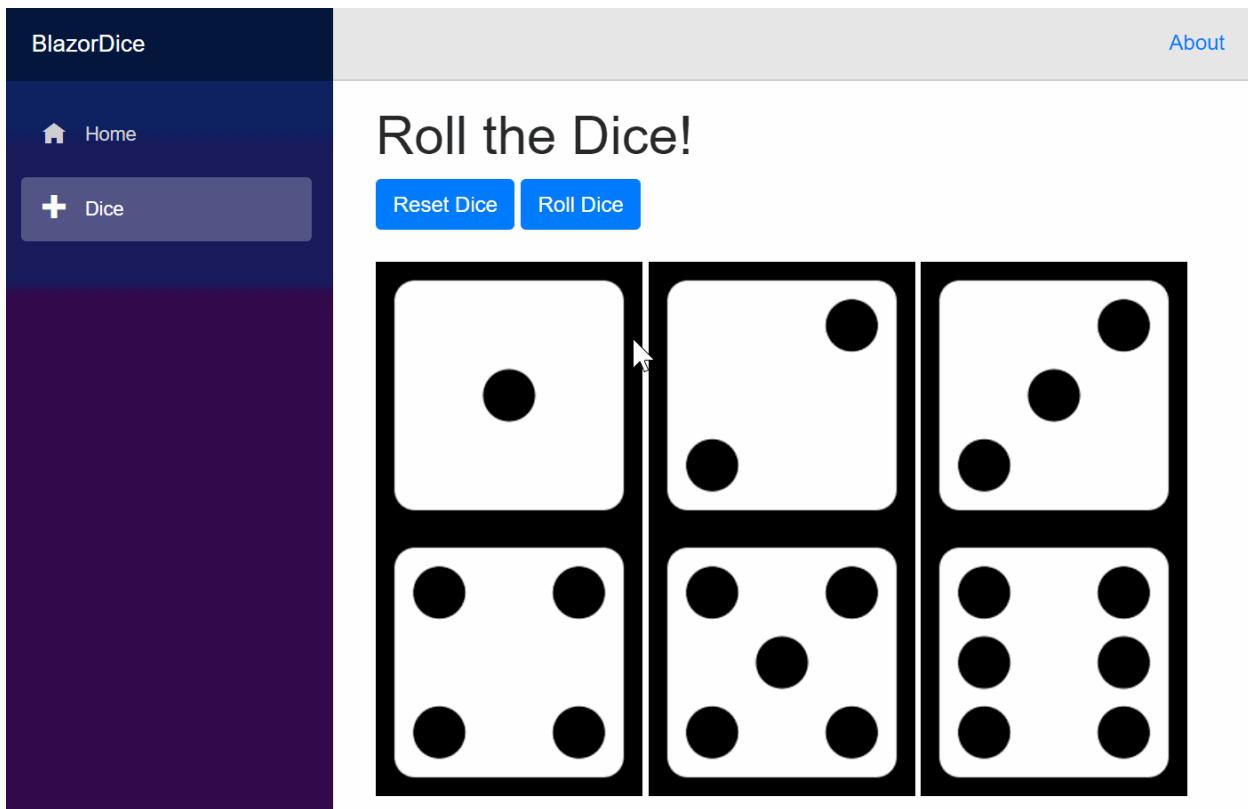
**NOTE:** *Server-side Blazor (aka Razor Components) allows you to run your Blazor app on the server, while using SignalR for the connection to the client, but we will focus on client-only Blazor in this article.*

To get started by yourself, follow the official instructions to set up your development environment and then build your first app. In the meantime, you may download the sample code from my GitHub repo.



Blazor projects on GitHub: <https://github.com/shahedc/BlazorDemos>

Specifically, take a look at the Blazor Dice project, which you can use to generate random numbers using dice graphics. The GIF below illustrates the web app in action!



GIF Source: <https://wakeupandcode.com/wp-content/uploads/2019/01/blazor-dice.gif>

## Entry Point and Configuration

Let's start with `Program.cs`, the entry point of your application. Just like any ASP .NET Core web application, there is a `Main` method that sets up the entry point. A quick call to `CreateHostBuilder()` in the same file ensures that two things will happen: The Blazor Web Assembly Host will call its own `CreateDefaultBuilder()` method (*similar to how it works in a typical ASP .NET Core web application*) and it will also call `UseBlazorStartup()` to identify the Startup class where the application is configured.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IWebAssemblyHostBuilder CreateHostBuilder(string[] args) =>
        BlazorWebAssemblyHost.CreateDefaultBuilder()
```

```
        .UseBlazorStartup<Startup>() ;  
    }
```

Note that the Startup class doesn't *have* to be called Startup, but you do have to tell your application what it's called. In the Startup.cs file, you will see the familiar **ConfigureServices()** and **Configure()** methods, but you won't need any of the regular MVC-related lines of code that set up the HTTP pipeline for an MVC (or Razor Pages) application. Instead, you just need a minimum of 1 line of code that adds the client side "App". (This is different for server-hosted apps.)

```
public class Startup  
{  
    public void ConfigureServices(IServiceCollection services)  
    {  
    }  
  
    public void Configure(IBlazorApplicationBuilder app)  
    {  
        app.AddComponent<App>("app");  
    }  
}
```

Note that the **Configure()** method takes in an app object of type **IBlazorApplicationBuilder**, unlike the usual **IApplicationBuilder** we see in regular ASP .NET Core web apps. When it adds the **App** component, it specifies the client-side app with the name "app" in double quotes.

**UPDATE:** In the above statement, I'm referring to "app" as "the client-side app". In the comments section, I explained to a reader (Jeff) how this refers to the HTML element in index.html, one of the 3 locations where you would have to change the name if you want to rename it. Another reader (Issac) pointed out that "app" should be described as "a DOM Element Selector Identifier for the element" in that HTML file, which Angular developers should also recognize. Issac is correct, as it refers to the <app> element in the index.html file.

**NAME CHANGES:** Issac also pointed out that "IBlazorApplicationBuilder has already become IComponentsApplicationBuilder". This refers to recent name changes on Jan 18, 2019. I will periodically make changes to the articles and code samples in this series. In the meantime, please refer to the following GitHub commit:

- [Commit] Rename IBlazorApplicationBuilder->IComponentsApplicationBuilder:  
<https://github.com/aspnet/AspNetCore/commit/c901cc069d644ce9b7a6d93c53895155df3ab27>  
6
- [PR] Naming and template updates: <https://github.com/aspnet/AspNetCore/pull/6843>

**NOTE:** There is an App.cshtml file in the project root that specifies the AppAssembly as a temporary measure, but the app config in this file is expected to move to Program.cs at a future date.

# Client-Side Application

This “app” is defined in index.html, which lives in the wwwroot folder along with other static files.

```
<html>
...
<body>
<app>Loading...</app>
<script src="_framework/blazor.webassembly.js"></script>
</body>
</html>
```

The HTML in the index page has two things worth noting: an **<app>** tag with some placeholder text, followed by a **<script>** tag with a reference to the Blazor JavaScript code that allows you to run your .NET code in the browser via WebAssembly.

Blazor’s page components can be used to create the UI and interaction, starting with one or more **@page** directives that define the page routes, followed by HTML elements and C# code to interact with those elements. Take, for example, the Index.cshtml file in the Pages folder:

```
@page "/"
<h1>Hello, Developers!</h1>
Get ready to roll the dice!
<NavLink class="nav-link" href="dice">
    <span>Try it now!</span>
</NavLink>
```

In the above code for Index.cshtml, the **@page** directive indicates that this is a Page (a type of Blazor component) and the “/” single slash character indicates that this is the root page for your application. A component can have multiple routes if desired.

The **<NavLink>** tag is a helper class that generates an **<a>** tag hyperlink for your browser. In this case, the href attribute points to another route “dice” in the application, defined in Dice.cshtml (shown below).

```

@page "/dice"

... HTML elements

@functions {
    // client-side C# code
}

```

In the above code for Dice.cshtml, once again the `@page` directive defines a route “/dice” that defines how you can arrive on this page. There are a couple of `<button>` elements that generate client-side HTML buttons, yet can trigger C# code with onclick handlers. The rest of the file contains C# code wrapped in a `@functions{}` block to be loaded in the browser.

The `@` symbols in the onclick handlers allow you to call C# code from the HTML buttons, to reset the dice and roll a new set of dice:

```

<button class="btn btn-primary" onclick="@ResetDice">Reset
Dice</button>
<button class="btn btn-primary" onclick="@RollDice">Roll Dice</button>

```

The `<div>` elements that follow contain placeholder images for six dice images, with `src` attributes defined by a C# array of strings. The dice images are named `dice1.png` through `dice6.png`.

```








```

The `@functions` block defines the aforementioned string array of dice images, along with other necessary variables.

```

int[] diceArray = new int[] { 1, 2, 3, 4, 5, 6 };
string[] diceImage = new string[6];
string dicePrefix = "/images/dice";
string diceSuffix = ".png";

```

## LifeCycle Methods

The `@functions` block continues with an (overridden) `OnInit()` method, which can be used to initialize anything when the page first loads. In this case, it calls `ResetDice()` to reset the dice images.

```

protected override void OnInit()
{

```

```
        ResetDice();  
    }
```

Blazor's Lifecycle methods for a component include the following:

- `OnInit()`: invoked after receiving initial params
- `OnInitAsync()`: async alternative to `OnInit()`;
- `OnParametersSet()`: called after receiving params from its parent, after `OnInit()`
- `OnParametersSetAsync()`: async alternative to `OnParametersSet()`;
- `OnAfterRender()`: called after each render
- `OnAfterRenderAsync()`: async alternative to `OnAfterRender()`
- `ShouldRender()`: used to suppress subsequent rendering of the component
- `StateHasChanged()`: called to indicate that state has changed, can be triggered manually

## Updating the UI

The rest of the C# code in Dice.cshtml includes methods to update the UI by changing the `src` property of each image. This is accomplished by the following:

```
void ResetDice()  
{  
    for (int diceIndex = 0; diceIndex < diceArray.Length; diceIndex++)  
    {  
        diceImage[diceIndex] = GetDicePath(diceIndex + 1);  
    }  
}  
  
void RollDice()  
{  
    Random rnd = new Random();  
    int[] randomDice = diceArray.OrderBy(x => rnd.Next()).ToArray();  
  
    for (int diceIndex = 0; diceIndex < diceArray.Length; diceIndex++)  
    {  
        diceImage[diceIndex] = GetDicePath(randomDice[diceIndex]);  
    }  
}
```

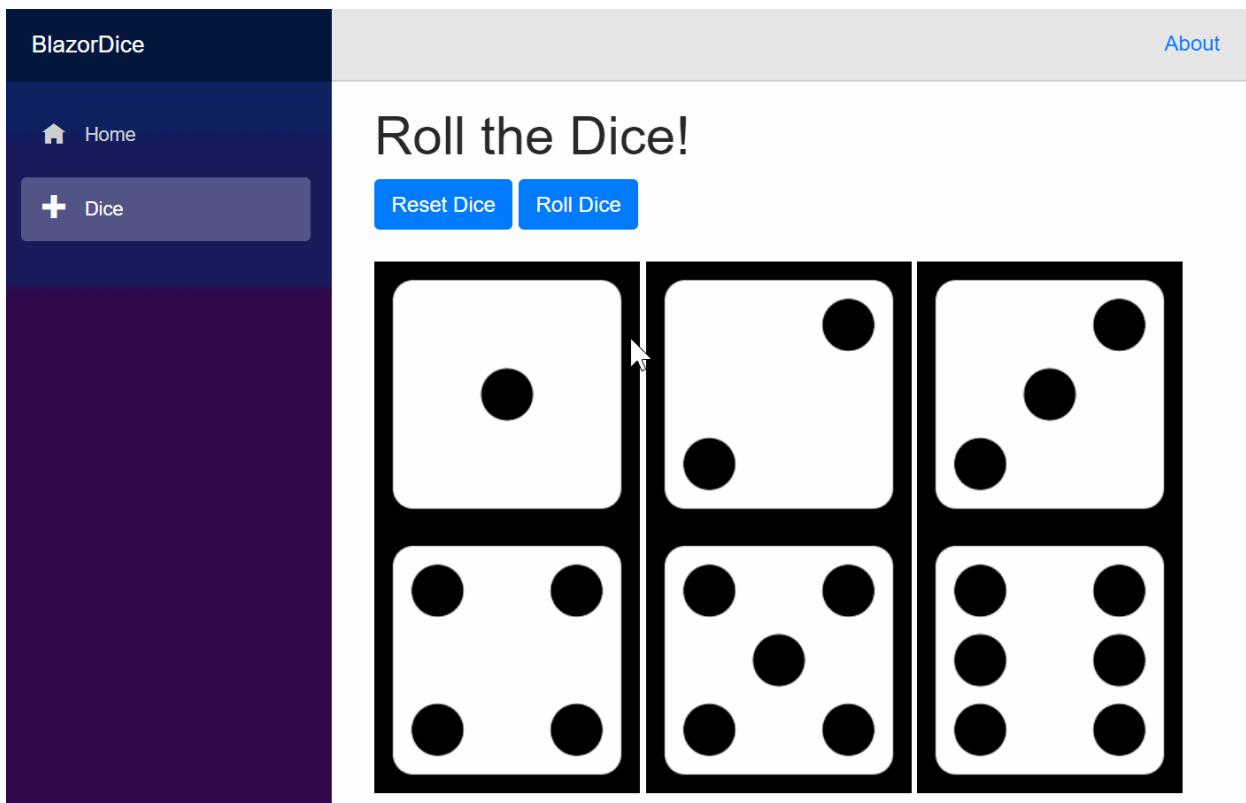
```
    }  
}
```

In the above C# code, the method **ResetDice()** loops through the predefined dice array and updates each **diceImage** element to update its respective src property. The **RollDice()** method generate a random series of dice values and then loops through the aforementioned array of dice to update each dice image to a random value from the random dice set.

Finally, we have the helper method **GetDicePath()**, which simply builds a path to a dice image by concatenating the path, filename and extension into a single string.

```
string GetDicePath(int diceValue)  
{  
    return $"{dicePrefix}{diceValue}{diceSuffix}";  
}
```

Once again, here is the end result, illustrated in a GIF:



GIF Source: <https://wakeupandcode.com/wp-content/uploads/2019/01/blazor-dice.gif>

# Next Steps

There is so much more to learn about this exciting new experimental framework. Blazor's reusable components can take various forms, including Pages (that you've just seen,) dialogs and forms.

In addition to client-side apps, you can also host Blazor apps on the server-side from within an ASP .NET Core web app. Converting your client-side app to a server-hosted app requires some minor code changes. Server-side Blazor (aka Razor Components) is expected to appear within the .NET Core 3.0 release in 2019.

Since everything is loaded in the browser via WebAssembly, you can use your browser's F12 Developer Tools to inspect and debug your C# code. You can also call existing JavaScript libraries using JavaScript interop.

# References

- Official Blazor Website: <https://blazor.net>
- Blazor 0.5.0 (July 2018)  
announcement: <https://blogs.msdn.microsoft.com/webdev/2018/07/25/blazor-0-5-0-experimental-release-now-available/>
- Blazor 0.6.0 (Oct 2018)  
announcement: <https://blogs.msdn.microsoft.com/webdev/2018/10/02/blazor-0-6-0-experimental-release-now-available/>
- Blazor 0.7.0 (Nov 2018)  
announcement: <https://blogs.msdn.microsoft.com/webdev/2018/11/15/blazor-0-7-0-experimental-release-now-available/>
- Blazor and more at Ignite (Nov 2018): <https://myignite.techcommunity.microsoft.com/sessions/65901>
- Learn Blazor: <https://learn-blazor.com/>
- Blazor on CodeDaze: <https://codedaze.io/blazor/>

- AdrienTorris' collection of awesome Blazor resources: <https://github.com/AdrienTorris/awesome-blazor>

From Ignite (Nov 2018): [https://www.youtube.com/watch?v=Yb-N0TmEq\\_Y](https://www.youtube.com/watch?v=Yb-N0TmEq_Y)

# Cookies and Consent in ASP .NET Core

By Shahed C on January 21, 2019

2 Replies

This is the **third** of a new series of posts on ASP .NET Core for 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



**A – Z of ASP .NET Core!**

## In this Article:

- C is for Cookies and Consent
- Browser Storage
- Partial Views for your cookie message
- Customizing your message
- Startup Configuration
- References

## C is for Cookies and Consent

In this article, we'll continue to look at the (in-progress) NetLearner application, which was generated using one of the standard ASP .NET Core web app project (2.2) templates. Specifically, let's take a look at how the template makes it very easy for you to store cookies and display a cookie policy.

**NOTE:** *The way cookies are handled in the project templates may change with each new release of ASP .NET Core.*

Unless you've been living under a rock in the past year or so, you've no doubt noticed all the GDPR-related emails and website popups all over the place. Whether or not you're required by law to disclose your cookie policies, it's good practice to reveal it to the end user so that they can choose to accept your cookies (or not).

## Browser Storage

As you probably know, cookies are attached to a specific browser installation and can be deleted by a user at any time. Some new developers may not be aware of where these cookies are actually stored.

Click F12 in your browser to view the Developer Tools to see cookies grouped by website/domain.

- In Edge/Firefox, expand Cookies under the Storage tab.
- In Chrome, expand Storage | Cookies under the Application tab .

See screenshots below for a couple of examples how AspNet.Consent is stored, along with a boolean Yes/No value:

The screenshot shows the Microsoft Edge DevTools interface. On the left is the main browser window displaying a sample ASP.NET Core application with a 'Welcome' page. On the right is the DevTools sidebar with several tabs: Elements, Console, Debugger, Network, and Cookies. The Cookies tab is active, showing a list of cookies for the domain <https://localhost:44389/>. The list includes:

Name	Value	Do...	Path	Exp...	Sa...	Secure	HttpOnly
.AspNet.Consent	yes	local...	/	Tue...	Lax	✓	

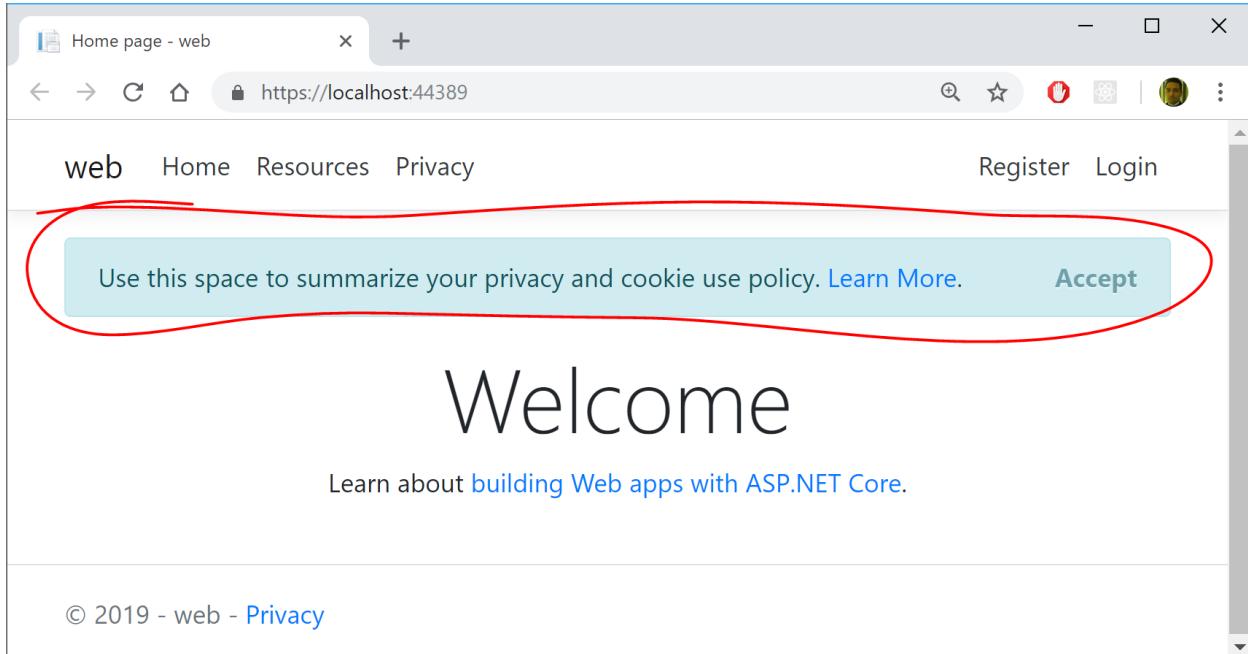
Cookies in Edge

The screenshot shows the Google Chrome DevTools interface. On the left is the main browser window displaying the same sample ASP.NET Core application. On the right is the DevTools sidebar. The Application tab is selected, showing the Storage section expanded. Under Storage, the Cookies section is listed, showing the .AspNet.Consent cookie for the domain <https://localhost:44389>.

Cookies in Chrome

# Partial Views for your cookie message

The first time you launch a new template-generated ASP .NET Core web app, you should see a cookie popup that appears on every page that can be dismissed by clicking Accept. Where does it come from? Let's explore the code to dig in a little further.



First, take a look at the `_CookieConsentPartial.cshtml` partial view, from which CSS class names and accessibility-friendly role attributes have been removed for brevity in the snippet below. For Razor Pages (in this example), this file should be in the **/Pages/Shared/** folder by default. For MVC, this file should be in the **/Views/Shared/** folder by default.

```
@using Microsoft.AspNetCore.Http.Features

@{
    var consentFeature =
        Context.Features.Get<ITrackingConsentFeature>();
    var showBanner = !consentFeature?.CanTrack ?? false;
    var cookieString = consentFeature?.CreateConsentCookie();
}

@if (showBanner)
{
    <div id="cookieConsent">
```

```

<!-- CUSTOMIZED MESSAGE IN COOKIE POPUP -->
<button type="button" data-dismiss="alert" data-cookie-
string="@cookieString">
    <span aria-hidden="true">Accept</span>
</button>
</div>
<script>
(function () {
    var button = document.querySelector("#cookieConsent button[data-
cookie-string]");
    button.addEventListener("click", function (event) {
        document.cookie = button.dataset.cookieString;
    }, false);
})();
</script>
}

```

This partial view has a combination of server-side C# code and client-side HTML/CSS/JavaScript code. First, let's examine the C# code at the very top:

1. The using statement at the top mentions the **Microsoft.AspNetCore.Http.Features** namespace, which is necessary to use **ITrackingConsentFeature**.
2. The local variable **consentFeature** is used to get an instance **ITrackingConsentFeature** (or null if not present).
3. The local variable **showBanner** is used to store the boolean result from the property **consentFeature.CanTrack** to check whether the user has consented or not.
4. The local variable **cookieString** is used to store the “cookie string” value of the created cookie after a quick call to **consentFeature.CreateConsentCookie()**.
5. The @if block that follows only gets executed if **showBanner** is set to true.

Next, let's examine the HTML that follows:

1. The **cookieConsent** <div> is used to store and display a customized message for the end user.
2. This <div> also displays an **Accept** <button> that dismisses the popup.
3. The **data-dismiss** attribute ensures that the modal popup is closed when you click on it. This feature is available because we are using Bootstrap in our project.
4. The data- attribute for “**data-cookie-string**” is set using the server-side variable value for **@cookieString**.

The full value for **cookieString** may look something like this, beginning with the **.AspNet.Consent** boolean value, followed by an expiration date.

```
.AspNet.Consent=yes; expires=Tue, 21 Jan 2020 01:00:47 GMT; path=/;
secure; samesite=lax
```

Finally, let's examine the JavaScript that follows within a **<script>** tag:

1. Within the **<script>** tag, an anonymous function is defined and invoked immediately, by ending it with **()**; after it's defined.
2. A **button** variable is defined to represent the HTML button, by using an appropriate **querySelector** to retrieve it from the DOM.
3. An eventListener is added to respond to the button's **onclick** event.
4. If accepted, a new cookie is created using the button's aforementioned **cookieString** value.

To use the partial view in your application, simply insert it into the `_Layout.cshtml` page that is used by all your pages. The partial view can be inserted above the call to **RenderBody()** as shown below.

```
<div class="container">
    <partial name="_CookieConsentPartial" />
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
</div>
```

In an MVC application, the partial view can be inserted the same way, using the **<partial>** tag helper.

## Customizing your message

You may have noticed that there is only an Accept option in the default cookie popup generated by the template's Partial View. This ensures that the only way to store a cookie with the user's consent is to click Accept in the popup.

You be wondering whether you should also display a Decline option in the cookie popup. But that would be a bad idea, because that would require you to store the user's "No" response in the cookie itself, thus

going against their wishes. If you wish to allow the user to *withdraw* consent at a later time, take a look at the `GrantConsent()` and `WithdrawConsent()` methods provided by `ITrackingConsentFeature`.

But you can still change the message in the cookie popup and your website's privacy policy. To change the cookie's displayed message, simply change the text that appears in the `_CookieConsentPartial.cshtml` partial view, within the `<div>` of the client-side HTML. In the excerpt shown in the previous section, this region is identified by the **<!-- CUSTOMIZED MESSAGE IN COOKIE POPUP -->** placeholder comment.

```
<div id="cookieConsent">
    <!-- CUSTOMIZED MESSAGE IN COOKIE POPUP -->
    <button type="button" data-dismiss="alert" data-cookie-
string="@cookieString">
        <span aria-hidden="true">Accept</span>
    </button>
</div>
```

Your message text is also a great place to provide a link to your website's privacy policy. In the Razor Pages template, the `<a>` link is generated using a tag helper shown below. The `/Privacy` path points to the `Privacy.cshtml` Razor page in the `/Pages` folder.

```
<a asp-page="/Privacy">Learn More</a>
```

In a similar MVC application, you would find the `Privacy.cshtml` view within the **/Views/Home/** folder, accessible via the Home controller's `Privacy()` action method. In the MVC template, the `<a>` is link is generated using the following tag helper:

```
<a asp-area="" asp-controller="Home" asp-action="Privacy">Learn
More</a>
```

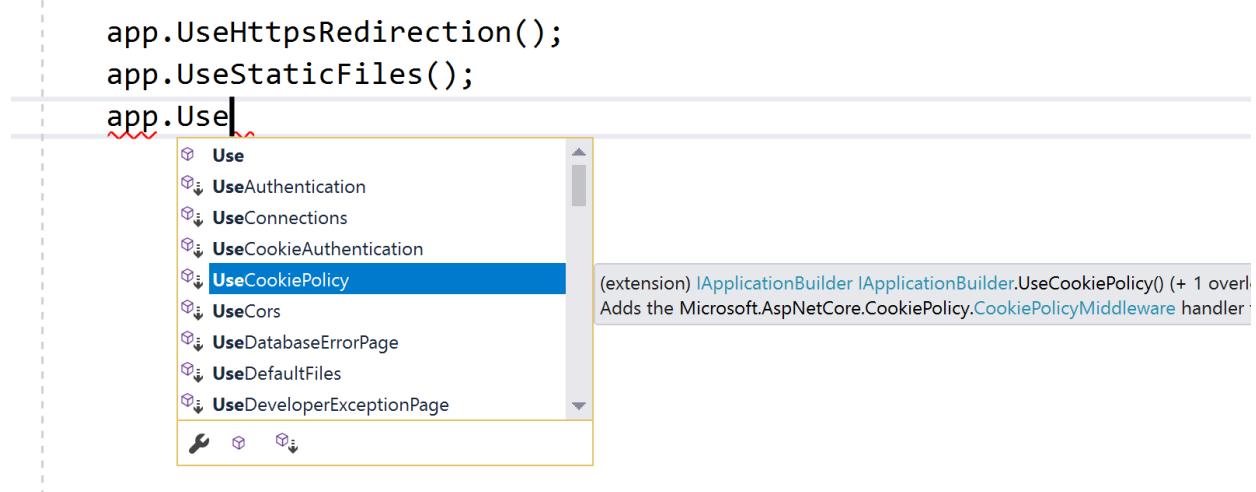
## Startup Configuration

None of the above would be possible without the necessary configuration. The cookie policy can be used by simply calling the extension method `app.UseCookiePolicy()` in the `Configure()` method of your `Startup.cs` file.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
{
    ...
    app.UseCookiePolicy();
    ...
    app.UseMvc();
}
```

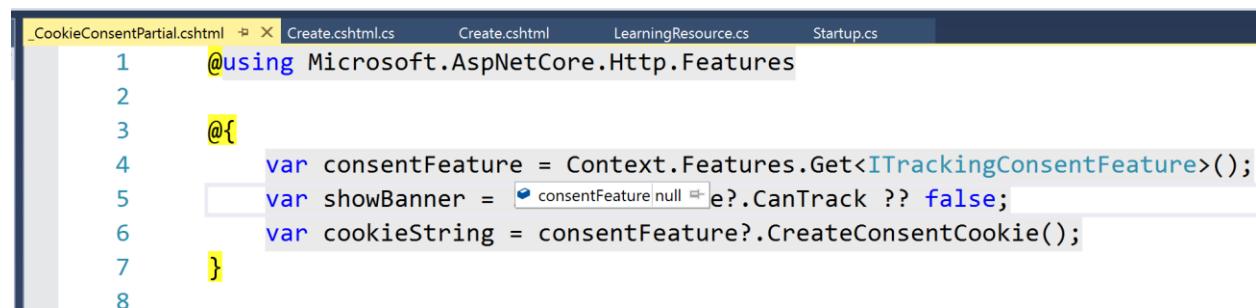
According to the official documentation, this “*Adds the Microsoft.AspNetCore.CookiePolicy.CookiePolicyMiddleware handler to the specified Microsoft.AspNetCore.Builder.IApplicationBuilder, which enables cookie policy capabilities.*”

The cool thing about ASP .NET Core middleware is that there are many IApplicationBuilder extension methods for the necessary Middleware components you may need to use. Instead of hunting down each Middleware component, you can simply type **app.Use** in the **Configure()** method to discover what is available for you to use.



If you remove the call to **app.UseCookiePolicy()**, this will cause the aforementioned **consentFeature** value to be set to null in the C# code of your cookie popup.

```
var consentFeature = Context.Features.Get<ITrackingConsentFeature>();
```



There is also some minimal configuration that happens in the **ConfigureServices()** method which is called *before* the **Configure()** method in your Startup.cs file.

```
public void ConfigureServices(IServiceCollection services)
{
```

```

services.Configure<CookiePolicyOptions>(options =>
{
    // This lambda determines whether user consent for non-essential
    cookies is needed for a given request.
    options.CheckConsentNeeded = context => true;
    options.MinimumSameSitePolicy = SameSiteMode.None;
}) ;
...
}

```

The above code does a couple of things:

1. As explained by the comment, the lambda (context => true) “determines whether user consent for non-essential cookies is needed for a given request” and then the **CheckConsentNeeded** boolean property for the options object is set to true or false.
2. The property **MinimumSameSitePolicy** is set to SameSiteMode.None, which is an enumerator with the following possible values:
  - None = 0
  - Lax = 1
  - Strict = 2

From the official documentation on cookie authentication, “*When set to SameSiteMode.None, the cookie header value isn’t set. Note that Cookie Policy Middleware might overwrite the value that you provide. To support OAuth authentication, the default value is SameSiteMode.Lax.*” This explains why even though the value is initially set to None in the **ConfigureServices()** method, using the Middleware in the **Configure()** method causes the “samesite” value to be set to “lax” in the cookiestring we observed earlier.

```
.AspNet.Consent=yes; expires=Tue, 21 Jan 2020 01:00:47 GMT; path=/;
secure; samesite=lax
```

## References

- General Data Protection Regulation (GDPR) support in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/security/gdpr>

- Use cookie authentication without ASP.NET Core Identity: <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/cookie>
- ITrackingConsentFeature Interface  
(Microsoft.AspNetCore.Http.Features): <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.features.itrackingconsentfeature?view=aspnetcore-2.2>
- HTMLElement.dataset: <https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dataset>
- Using the alert role: [https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA\\_Techniques/Using\\_the\\_alert\\_role](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques/Using_the_alert_role)
- HTML DOM querySelector()  
Method: [https://www.w3schools.com/jsref/met\\_document\\_queryselector.asp](https://www.w3schools.com/jsref/met_document_queryselector.asp)

# Deploying ASP .NET Core to Azure App Service

By Shahed C on January 28, 2019

8 Replies

This is the **fourth** of a new series of posts on ASP .NET Core for 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

- D is for Deploying to Azure App Service
- Right-Click Publish (aka...)
- Web Apps in the Azure Portal
- Runtime Options
- Deployment Center
- GitHub Repos
- CLI Commands
- References

Before you begin, make sure you sign in to Azure, or create a new trial account first to follow along.

# D is for Deploying to Azure App Service

In this article, we'll explore several options for deploying an ASP .NET Core web app to Azure App Service in the cloud. From the infamous Right-Click-Publish to fully automated CI/CD, you'll learn about the latest Deployment Center option in the Azure Portal for App Service for web apps.

**NOTE:** If you're looking for information on deploying to Docker or Kubernetes, please check out the following docs instead:

- Host and deploy ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy>
- Deploy to Azure Kubernetes Service: <https://docs.microsoft.com/en-us/azure/devops-project/azure-devops-project-aks>
- Host ASP.NET Core in Docker containers: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/docker>
- Dockerize a .NET Core application: <https://docs.docker.com/engine/examples/dotnetcore/>

## Right-Click Publish (aka Friends Don't Let Friends Right-Click Publish)

If you've made it this far, you may be thinking one of the following:

- a. "Hey, this is how I deploy my web apps right now!"  
or  
b. "Hey wait a minute, I've heard that you should never do this!"

Well, there is a time and place for right-click publish. There have been many debates on this, so I won't go into the details, but here are some resources for you to see what others are saying:

- [MSDN] **When should you right click publish**: <https://blogs.msdn.microsoft.com/webdev/2018/11/09/when-should-you-right-click-publish/>
- **Friends don't let friends right-click publish**[by Damian Brady]: <https://damianbrady.com.au/2018/02/01/friends-dont-let-friends-right-click-publish/>
- **In the wild** [by Geoffrey Huntley]: <https://rightclickpublish.com/>

Tweet from Geoffrey Huntley: <https://twitter.com/GeoffreyHuntley/status/994345821276536832>

And we are live!

 <https://t.co/5Q681RVVuu> 

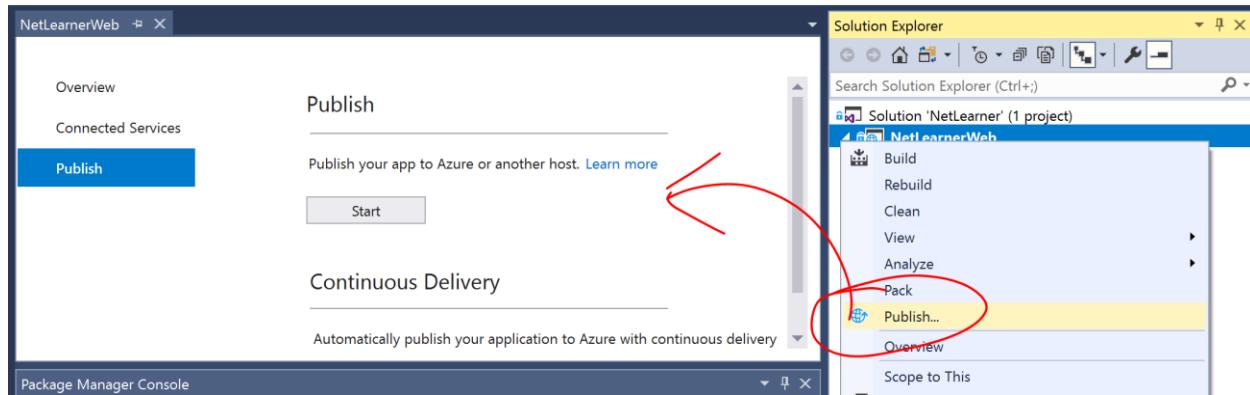
 PUSH TO MASTER TO DEPLOY  <pic.twitter.com/vxSG9MXgJN>

— geoffrey huntley (@GeoffreyHuntley) May 9, 2018

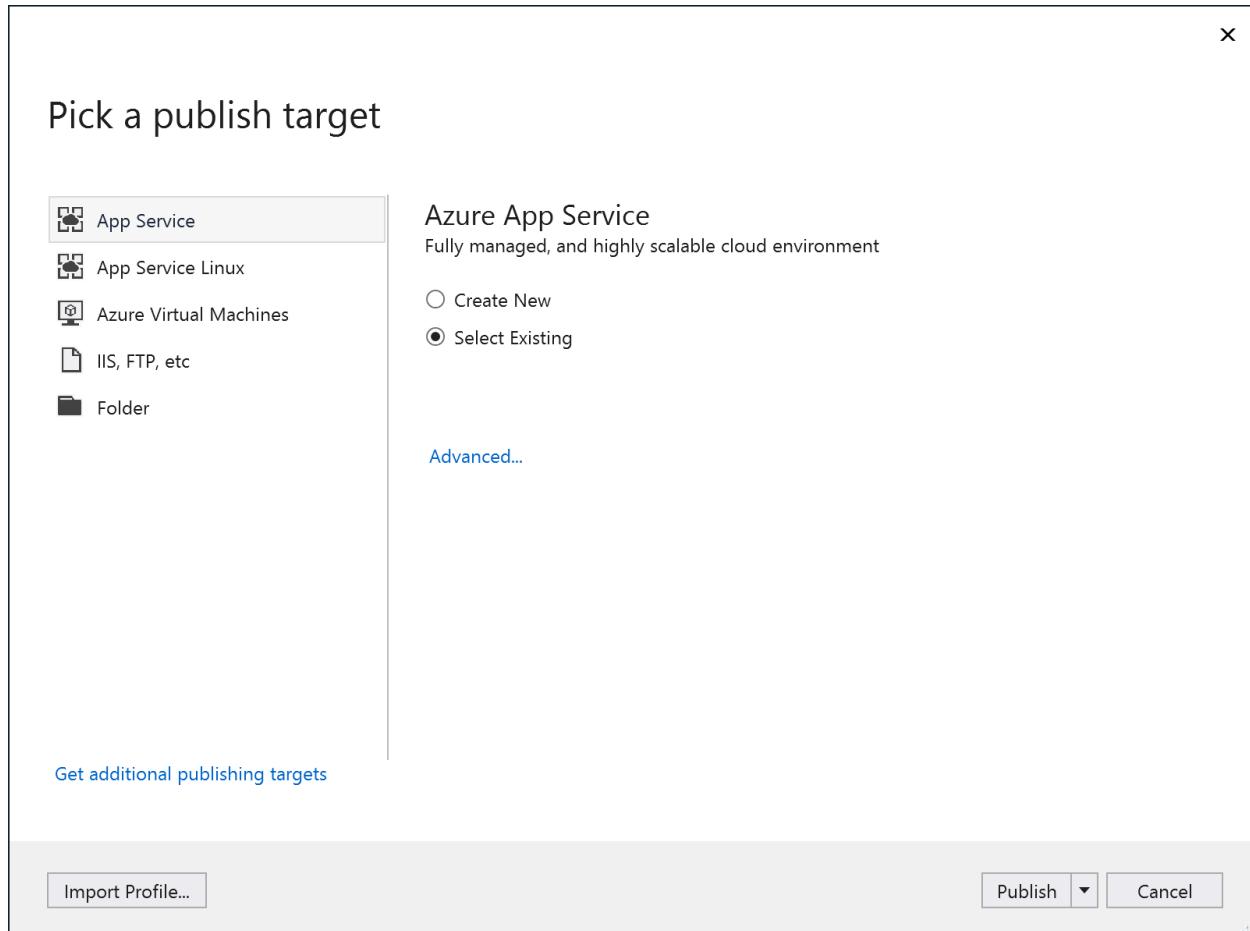
So, what's a web developer to do? To quote from the aforementioned MSDN article, "*Continuing with the theme of prototyping and experimenting, right click publish is the perfect way for existing Visual Studio customers to evaluate Azure App Service (PAAS). By following the right click publish flow you get the opportunity to provision new instances in Azure and publish your application to them without leaving Visual Studio.*"

In other words, you can use this approach for a quick test or demo, as shown in the screenshots below for Visual Studio.

1. Right-click your ASP .NET Core web app project in Solution Explorer and select Publish.
2. Click the Start button on the screen that appears and follow the onscreen instructions.
3. Ensure that you're logged in to the correct Azure subscription account you want to publish to.



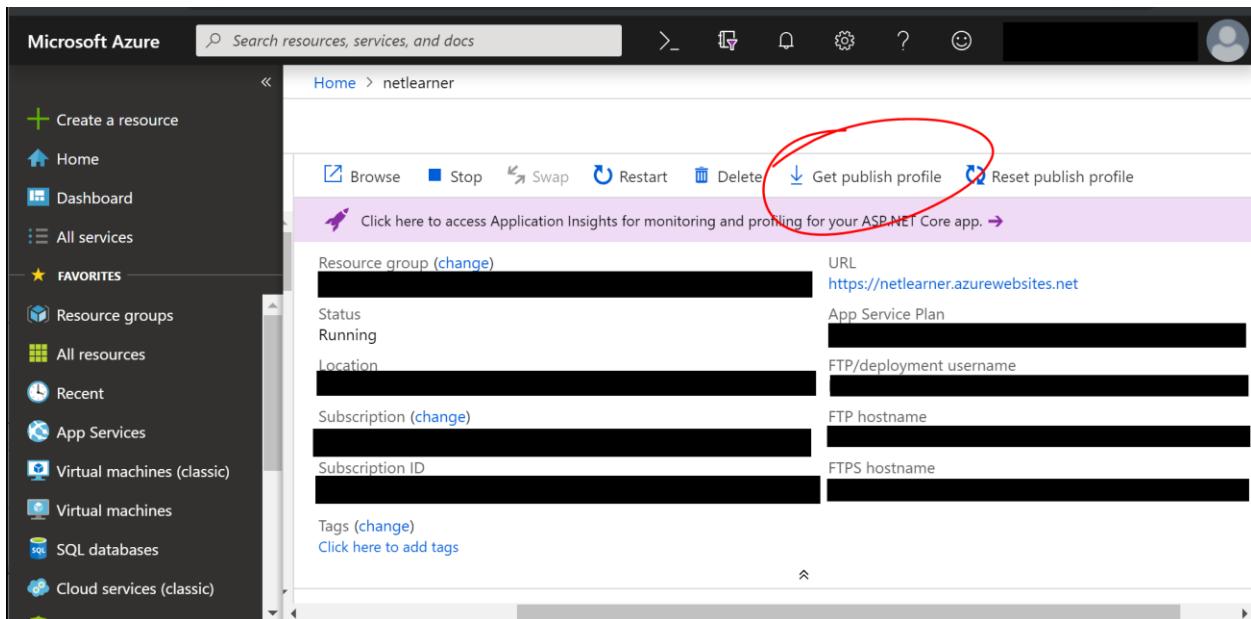
Right-click Publish your project from Solution Explorer



Follow the onscreen instructions

# Web Apps in the Azure Portal

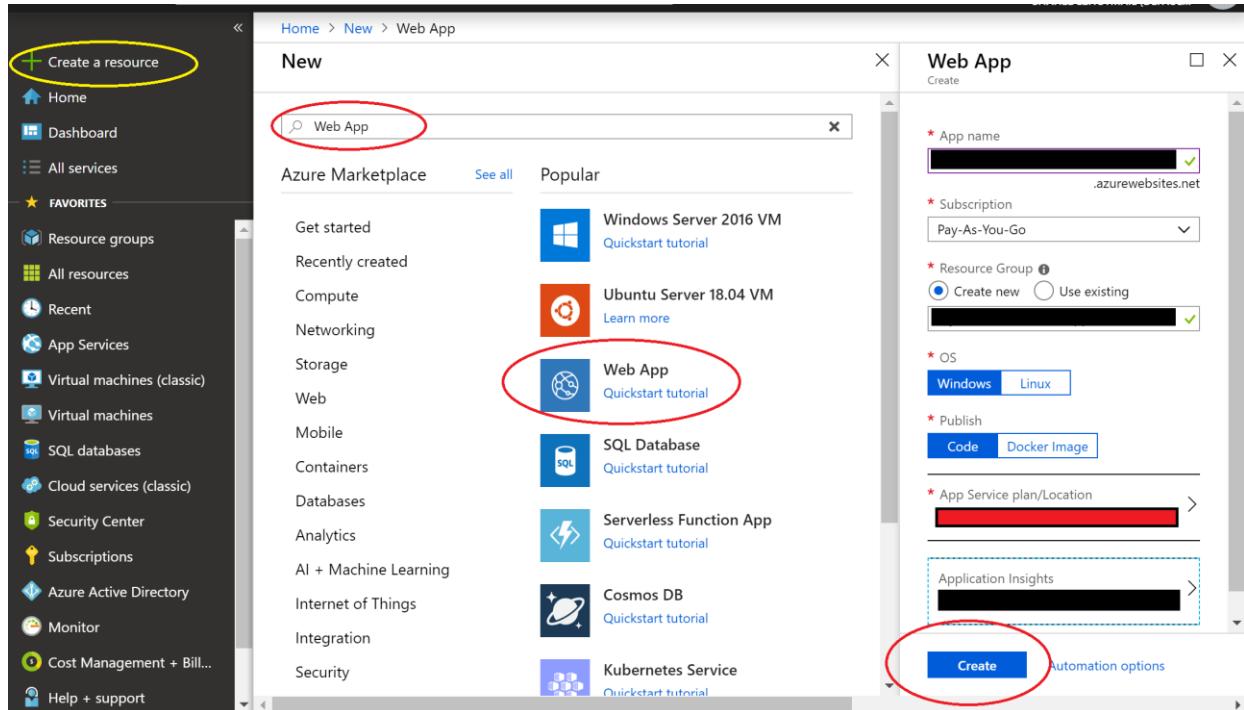
In the screenshot above, you may notice an option to “Import Profile” using the button on the lower left. This allows you to import a Web App profile file that was generated by exporting it from an *existing* Azure Web App. To grab this profile file, simply navigate to your existing Web App in the Azure Portal and click on “Get publish profile” in the top toolbar of your Web App, shown below:



If you want to create a new Web App in Azure starting with the Azure Portal, follow the instructions below:

1. Log in to the Azure at <https://portal.azure.com>
2. On the top left, click + Create a resource
3. Select “Web App” or search for it if you don’t see it.
4. Enter/select the necessary values:
  - App name (enter a unique name)

- Subscription (select a subscription)
  - Resource Group (create or use existing to group resources logically)
  - OS (Windows or Linux)
  - Publish (Code or Docker Image)
  - App Service Plan (create or use existing to set location and pricing tier)
  - Application Insights (now enabled by default)
5. Click the Create button to complete the creation of your new Web App.



New Web App in Azure

Now you can deploy to this Web App using any method you choose.

## Runtime Options

If you like to stay ahead of ASP .NET Core releases, you may be using a pre-release version of the runtime. As of this writing, the latest stable version of ASP .NET Core is version 2.2, but there is a preview of version 3.0 available. Fortunately, Azure App Service also has an option to install an Extension for preview runtimes.

To find the proper runtime:

1. Navigate to your Web App in the Azure Portal.
2. Click on Extensions under Development Tools.
3. Click + Add to add a new extension.
4. Choose an extension to configure required settings.
5. Accept the legal terms and complete the installation.

The screenshot shows the Microsoft Azure portal interface. On the left, the navigation bar includes 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES' (with 'netlearner' listed), 'Resource groups', 'All resources', and 'Recent'. The main area shows the URL 'Home > netlearner - Extensions > Add extension > Choose extension'. A modal window titled 'Add extension' contains fields for 'Choose Extension' (set to 'Configure required settings') and 'Legal Terms' (with a link to 'Accept legal terms'). To the right, a separate modal window titled 'Choose extension' lists four options: '.NET ASP.NET Core 2.2 (x64) R...', '.NET ASP.NET Core 2.2 (x86) R...', '.NET ASP.NET Core 3.0 (x64) R...', and '.NET ASP.NET Core Logging E...'. The 'ASP.NET Core 2.2 (x64) R...' item is highlighted with a blue dashed border.

Your options may include both 32-bit (x86) and 64-bit (x64) versions of the ASP .NET Core runtime and any preview versions of future releases. When planning ahead for multiple environments, you also have the option to deploy to Deployments Slots. This feature is available in **Standard**, **Premium** or **Isolated** App Service Plan tiers and will be covered in a future blog post in this series.

If you're interested in Deployment Slots right now, check out the official docs at:

- Set up staging environments for web apps in Azure App Service: <https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots>

## Deployment Center

In the list of features for your Web App, you will find an option to open up the new Deployment Center. Note that this is replacing the old Deployment Options (now labeled as Classic). Let's go over each of these options:

1. Azure Repos
2. Github
3. Bitbucket
4. Local Git
5. OneDrive
6. Dropbox

The screenshot shows the Azure App Service Deployment Center interface. On the left, a sidebar lists navigation items such as Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Deployment (Quickstart, Deployment slots (Preview), Deployment slots, Deployment options (Classic), Deployment Center), Settings (Application settings, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains, SSL settings), and Home.

The main area features a title "Deployment Center" with a gear icon. Below it is a sub-header: "App Service Deployment Center enables you to choose the location of your code as well as options for build and deployment to the cloud. [Learn more](#)".

A horizontal navigation bar at the top of the main content area has three items: "SOURCE CONTROL" (highlighted with a blue circle containing the number 1), "BUILD PROVIDER" (highlighted with a grey circle containing the number 2), and "CONFIGURE" (highlighted with a grey circle containing the number 3).

The "SOURCE CONTROL" section contains six cards:

- Azure Repos**: Configure continuous integration with an Azure Repo, part of Azure DevOps Services (formerly known as VSTS). Status: Not Authorized.
- Github**: Configure continuous integration with a GitHub repo. Status: Not Authorized.
- Bitbucket**: Configure continuous integration with a Bitbucket repo. Status: Not Authorized.
- Local Git**: Deploy from a local Git repo. Status: Not Authorized.
- OneDrive**: Sync content from a OneDrive cloud folder. Status: Not Authorized.
- Dropbox**: Sync content from a Dropbox cloud folder. Status: Not Authorized.

## Deployment Center for Web App

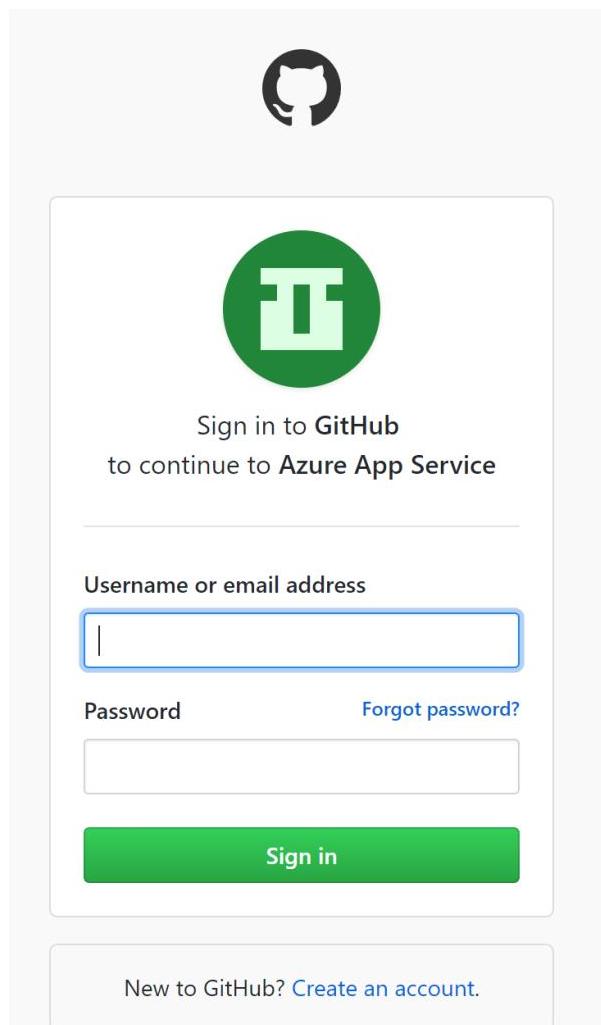
If you choose **Azure Repos**, you can set up your web app's CI (Continuous Integration) system with an Azure Repo, which is part of Microsoft's Azure DevOps services (formerly known as VSTS, aka Visual Studio Team Services). You will have options for using App Service as a Kudu build server or Azure Pipelines as your CI build system.



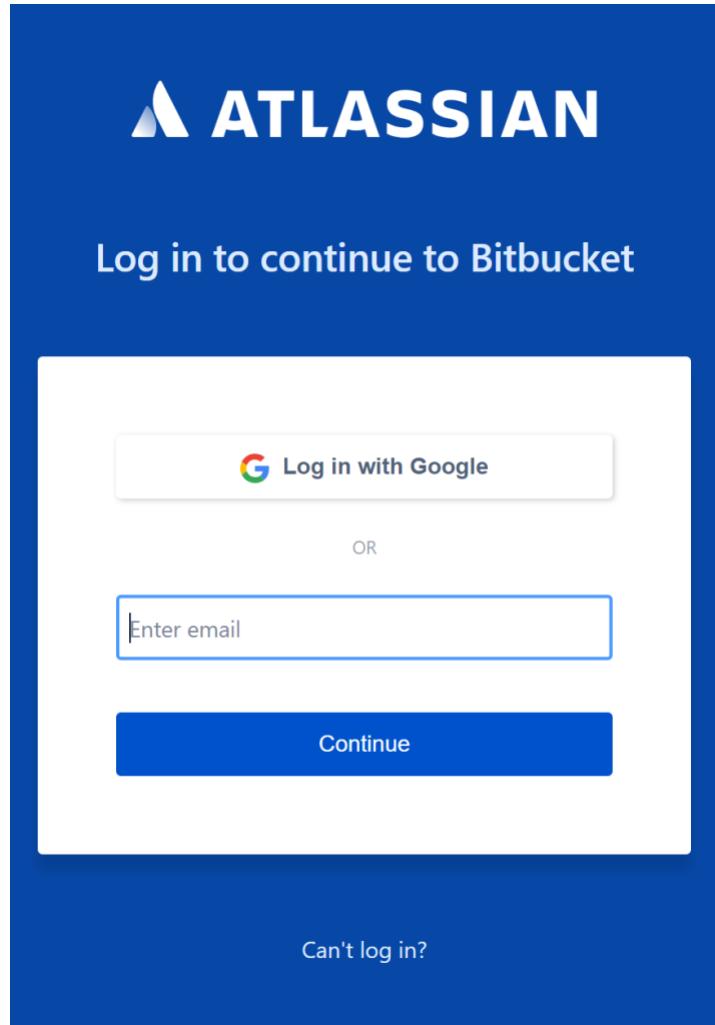
 App Service Kudu build server	 Azure Pipelines (Preview)
Use App Service as the build server. The App Service Kudu engine will automatically build your code during deployment when applicable with no additional configuration required.	Configure a robust deployment pipeline for your application using Azure Pipelines, part of Azure DevOps Services (formerly known as VSTS). The pipeline builds, runs load tests and deploys to staging slot and then to production.

Options for Azure Repos: App Service Kudu build server OR Azure Pipelines

If you choose **Github** or **BitBucket** or even a **local Git account**, you'll have the ability to authorize that account to publish a specific repo, every time a developer pushes their code.

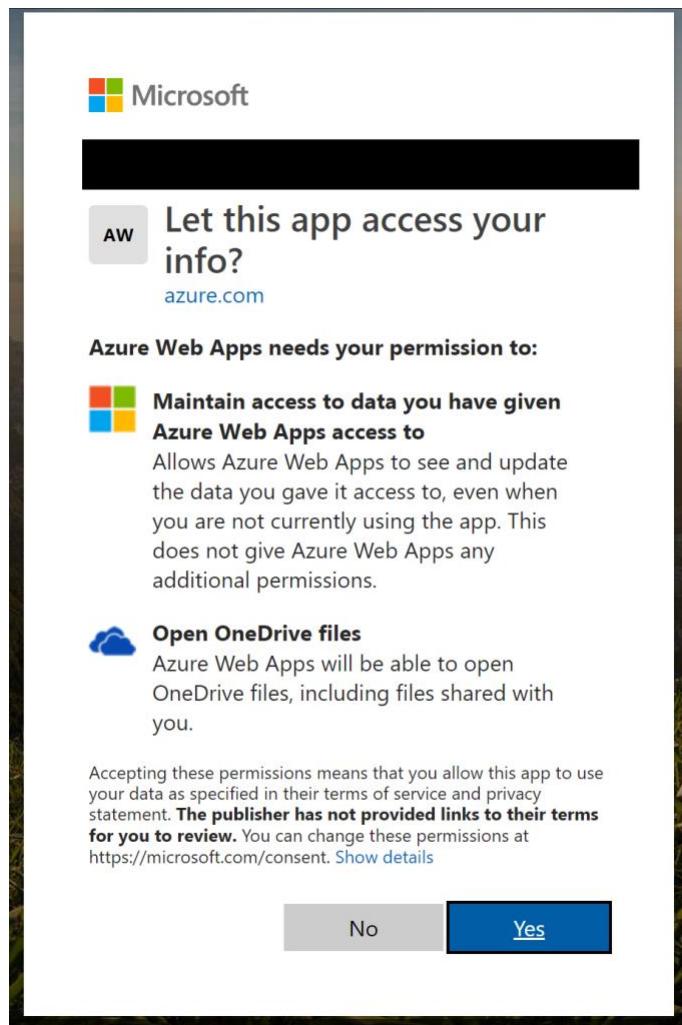


GitHub option



BitBucket option

If you choose **OneDrive** or **DropBox**, you'll have ability to authorize your App Service to pick up files deployed to a shared folder in either location.



OneDrive option



Sign in to Dropbox to link with Windows Azure

[Sign in with Google](#)

or

Email

Password

This page is protected by reCAPTCHA, and subject to the Google [Privacy Policy](#) and [Terms of Service](#).

[Forgot your password?](#) [Sign in](#)

DropBox option

To learn more about **Azure Repos** and **Azure Pipelines**, check out the official docs:

- Azure Pipelines: <https://docs.microsoft.com/en-us/azure/devops/pipelines>
- Overview: <https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started>
- Get Started: <https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started-yaml>
- Via Portal: <https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started-azure-devops-project>

# GitHub Repos (new FREE option!)

If you've been using GitHub for public open-source projects or private projects on paid accounts, now is a great time to create private repositories for free! As of January 2019, GitHub is now offering free unlimited private repos, limited to 3 collaborators. This new free option comes with issue/bug tracking and project management as well.

For more information on GitHub pricing, check out their official pricing page:

- GitHub Pricing: <https://github.com/pricing>

Free	Pro
<b>\$0</b>	<b>\$7</b>
Per month	Per month
The basics of GitHub for every developer	Pro tools for developers with advanced requirements
<ul style="list-style-type: none"><li>∞ Unlimited public repositories</li><li>∞ Unlimited private repositories <b>NEW</b></li><li>✓ 3 collaborators for private repositories</li><li>✓ Issues and bug tracking</li><li>✓ Project management</li></ul>	<ul style="list-style-type: none"><li>∞ Unlimited public repositories</li><li>∞ Unlimited private repositories</li><li>∞ Unlimited collaborators</li><li>✓ Issues and bug tracking</li><li>✓ Project management</li><li>✓ <a href="#">Advanced tools and insights</a></li></ul>

## GitHub Repo Pricing for Individuals

Now you can easily set up your starter projects in a private GitHub repository and take advantage of the aforementioned CI/CD setup without having to choose between paying a GitHub fee or making all your repos public.

# CLI Commands

If you wish to publish to Azure App service using CLI (Command Line Interface) Commands, you may use the following commands, where you can choose a name for your Web App, Resource Group, App Service Plan, etc. Single-letter flags are usually preceded by a single hyphen, while flags spelled out with completed words are usually preceded by two hyphens.

First, install the Azure CLI in case you don't have it already:

- Install the Azure CLI: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli>

Authenticate yourself:

```
> az login
```

Create a new resource group:

```
> az group create -l <LOCATION> -n <RSG>
> az group create --location <LOCATION> --name <RSG>
```

Create a new App Service Plan, where <SKUCODE> sku may be F1 or FREE, etc

```
> az appservice plan create -g <RSG> -n <ASP> --sku <SKUCODE>
> az appservice plan create --resource-group <RSG> --name <ASP> --sku
<SKUCODE>
```

From the documentation, the SKU Codes include: F1(Free), D1(Shared), B1(Basic Small), B2(Basic Medium), B3(Basic Large), S1(Standard Small), P1(Premium Small), P1V2(Premium V2 Small), PC2(Premium Container Small), PC3 (Premium Container Medium), PC4 (Premium Container Large).

Create a new Web App within a Resource Group, attached to an App Service Plan:

```
> az webapp create -g <RSG> -p <ASP> -n <APP>
> az webapp create --resource-group <RSG> --plan <ASP> --name <APP>
```

The above command should create a web app available at the following URL:

- <http://<APP>.azurewebsites.net>

To push your commits to a Git Repo and configure for App Service Deployment, use the following CLI commands:

Create a new Git repo or reinitialize an existing one:

```
> git init
```

Add existing files to the index:

```
> git add .
```

Commit your changes with a commit message:

```
> git commit -m "<COMMIT MESSAGE>"
```

Set your FTP credentials and Git deployment credentials for your Web App:

```
> az webapp deployment user set --user-name <USER>
```

Configure an endpoint and add it as a git remote.

```
> az webapp deployment source config-local-git -g <RSG> -n <APP> --out tsv  
> az webapp deployment source config-local-git --resource-group <RSG>  
--name <APP> --out tsv > git remote add azure <GIT URL>
```

The value for GIT URL is the value of the Git remote, e.g.

- <https://<USER>@<APP>.scm.azurewebsites.net/<APP>.git>

Finally, push to the Azure remote to deploy your Web App:

```
> git push azure master
```

For more information on CLI Commands for Git and Azure App Service, check out the official docs:

- Sign in with Azure CLI: <https://docs.microsoft.com/en-us/cli/azure/authenticate-azure-cli>
- az appservice plan: <https://docs.microsoft.com/en-us/cli/azure/appservice/plan>
- Deploy from local Git repo: <https://docs.microsoft.com/en-us/azure/app-service/deploy-local-git>
- az webapp deployment user: <https://docs.microsoft.com/en-us/cli/azure/webapp/deployment/user>

- Create an app with deployment from GitHub: <https://docs.microsoft.com/en-us/azure/app-service/scripts/cli-deploy-github>
- az webapp deployment source: <https://docs.microsoft.com/en-us/cli/azure/webapp/deployment/source?view=azure-cli-latest>

## References

- Verify ASP.NET Core on App Service: <https://aspnetcoreon.azurewebsites.net/>
- Deploy a web app to App Services – Azure Pipelines: <https://docs.microsoft.com/en-us/azure/devops/pipelines/apps/cd/deploy-webdeploy-webapps?view=vsts>
- Continuous deployment – Azure App Service: <https://docs.microsoft.com/en-us/azure/devops/pipelines/apps/cd/deploy-webdeploy-webapps?view=vsts>
- CI/CD for Azure Web Apps: <https://azure.microsoft.com/en-us/solutions/architecture/azure-devops-continuous-integration-and-continuous-deployment-for-azure-web-apps/>
- Azure Pipelines Overview: <https://docs.microsoft.com/en-us/azure/devops/pipelines/?view=vsts>
- Get started with Azure Pipelines: <https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/index?view=vsts>
- Continuous deployment: <https://docs.microsoft.com/en-us/azure/app-service/deploy-continuous-deployment>

# EF Core Relationships in ASP .NET Core

By Shahed C on February 4, 2019

1 Reply

This is the **fifth** of a new series of posts on ASP .NET Core for 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

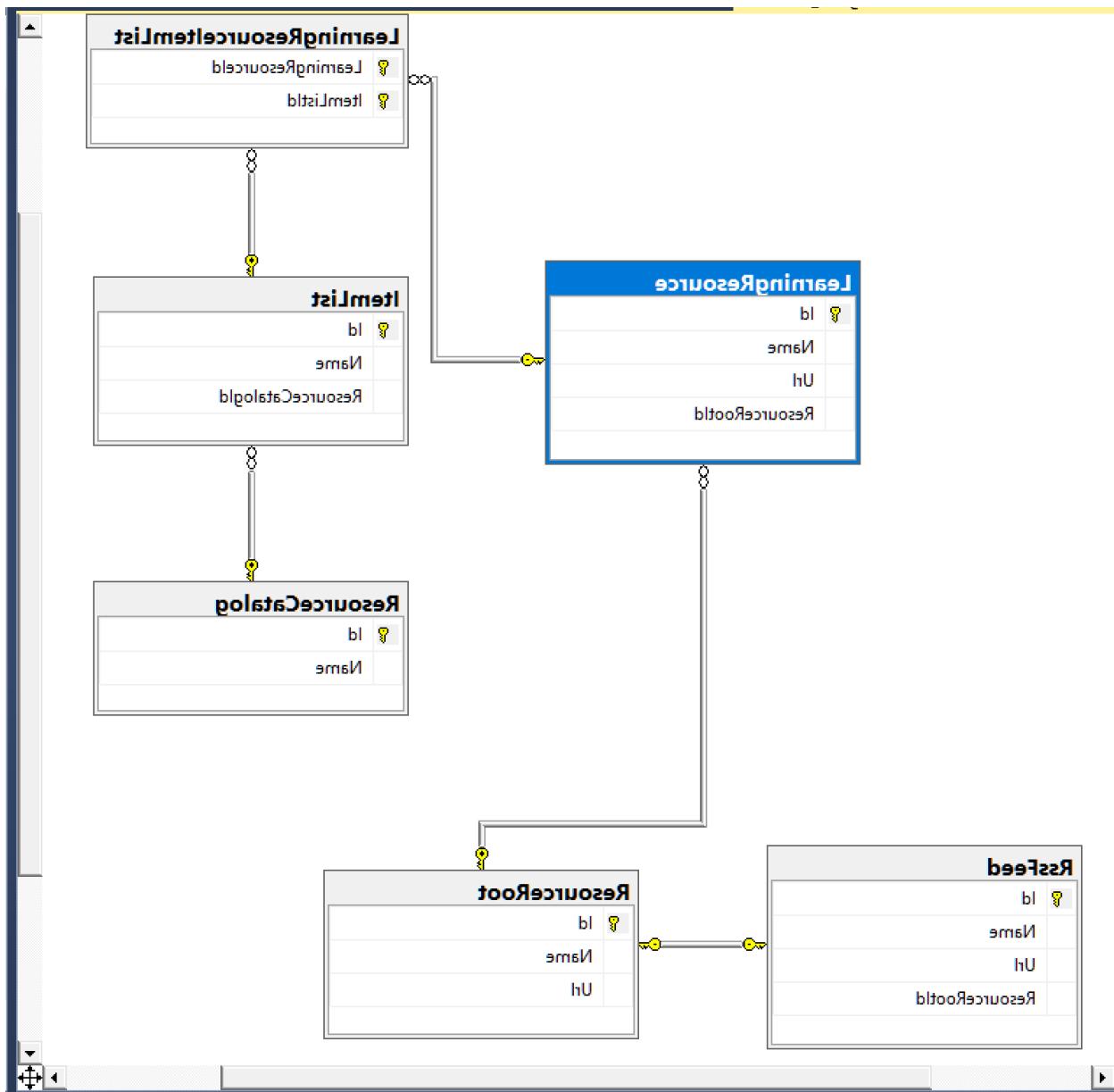
- E is for EF Core Relationships
- Classes and Relationships
- One to One
- One to Many (Example 1)
- One to Many (Example 2)
- Many to Many
- References

# E is for EF Core Relationships

In my 2018 series, we covered EF Core Migrations to explain how to add, remove and apply Entity Framework Core Migrations in an ASP .NET Core web application project. In this article, we'll continue to look at the NetLearner project, to identify entities represented by C# model classes and the relationships between them.

- NetLearner on GitHub: <https://github.com/shahedc/NetLearner>

**NOTE:** Please note that NetLearner is a work in progress as of this writing, so its code is subject to change. The UI still needs work (and will be updated at a later date) but the current version has the following models with the relationships shown below:



NetLearner database diagram

## Classes and Relationships

The heart of the application is the **LearningResource** class. This represents any online learning resource, such as a blog post, single video, podcast episode, ebook, etc that can be accessed with a unique URL.

```

public class LearningResource : InternetResource
{
    public List<LearningResourceItemList> LearningResourceItemLists
    {
        get; set;
    }
}

```

The abstract class **InternetResource** defines the common properties (e.g. Id, Name and Url) found in any Internet resource, and is also used by other classes **ResourceRoot** and **RssFeed**.

```

public abstract class InternetResource
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
}

```

The **ResourceRoot** class represents a root-level resource (e.g. a blog home or a podcast website) while the **RssFeed** class represents the RSS Feed for an online resource.

```

public class ResourceRoot: InternetResource
{
    public RssFeed RssFeed { get; set; }
    public List<LearningResource> LearningResources { get; set; }
} public class RssFeed: InternetResource
{
    public int ResourceRootId { get; set; }
}

```

The **ItemList** class represents a logical container for learning resources in the system. It is literally a list of items, where the items are your learning resources.

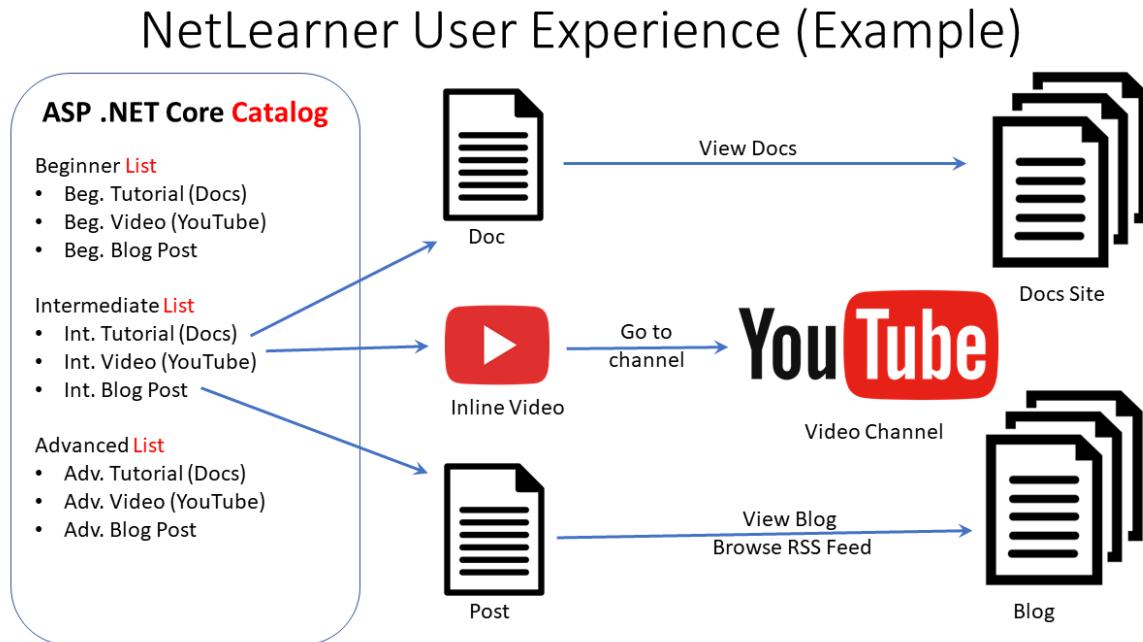
```

public class ItemList
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<LearningResourceItemList> LearningResourceItemLists
    {
        get; set;
    }
}

```

At this point, you may have noticed both the **LearningResource** and **ItemList** classes contain a **List<T>** property of **LearningResourceItemList**. If you browse the database diagram, you will notice that this table appears as a connection between the two aforementioned tables, to establish a many-to-many relationship. (more on this later)

The following diagram shows an example of how the a **LearningResource** is a part of a list (which is a part of a **ResourceCatalog**), while each **LearningResource** also has a **ResourceRoot**.



NetLearner example

## One to One

Having looked through the above entities and relationships, we can see that each ResourceRoot has an RssFeed. This is an example of a 1-to-1 relationship. For example:

- Resource Root = Wake Up and Code! blog site
- Rss Feed = RSS Feed for blog site

In the two classes, we see the following code:

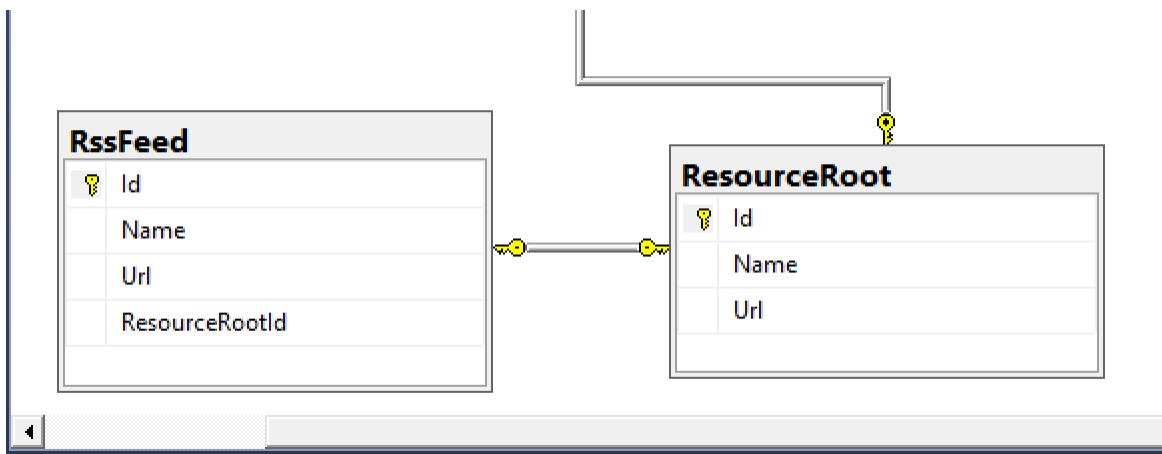
```
public class ResourceRoot: InternetResource
{
```

```

    public RssFeed RssFeed { get; set; }
    public List<LearningResource> LearningResources { get; set; }
} public class RssFeed: InternetResource
{
    public int ResourceRootId { get; set; }
}

```

Each Resource Root has a corresponding Rss Feed, so the **ResourceRoot** class has a property for **RssFeed**. That's pretty simple. But in the **RssFeed** class, you don't need a property pointing back to the **ResourceRoot**. In fact, all you need is a **ResourceRootId** property. EF Core will ensure that **ResourceRoot.Id** points to **RssFeed.ResourceRootId** in the database.



One to One Relationships

If you're wondering how these two classes got their **Id**, **Name** and **Url** fields, you may recall that they are both derived from a common abstract parent class (**InternetResource**) that define all this fields for reuse. **But wait a second...** why doesn't this parent class appear in the database? That's because we don't need it in the database and have intentionally omitted it from the list of **DbSet<>** definitions in the DB Context for the application, found in **ApplicationDbContext.cs**:

```

public class ApplicationDbContext : IdentityDbContext
{
    ...
    public DbSet<ItemList> ItemList { get; set; }
    public DbSet<LearningResource> LearningResource { get; set; }
    public DbSet<ResourceCatalog> ResourceCatalog { get; set; }
    public DbSet<ResourceRoot> ResourceRoot { get; set; }
    public DbSet<RssFeed> RssFeed { get; set; }
    ...
}

```

Another way of looking at the One-to-One relationship is to view the constraints of each database entity in the visuals below. Note that both tables have an Id field that is a Primary Key (inferred by EF Core) while the **RssFeed** table also has a Foreign Key for the **ResourceRootId** field used for the constraint in the relationship.

dbo.RssFeed [Design]			
Update   Script File: dbo.RssFeed.sql			
	Name	Data Type	Allow Nulls
1	Id	int	<input type="checkbox"/>
2	Name	nvarchar(MAX)	<input checked="" type="checkbox"/>
3	Url	nvarchar(MAX)	<input checked="" type="checkbox"/>
4	ResourceRootId	int	<input type="checkbox"/>
5			<input type="checkbox"/>

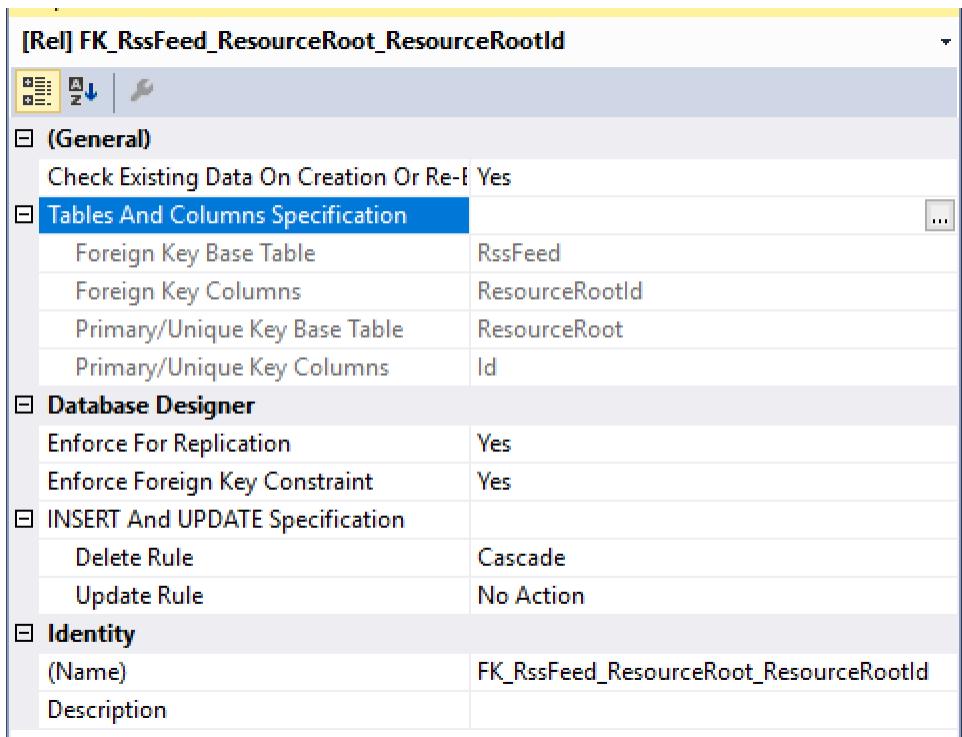
- ▲ **Keys** (1)  
PK\_RssFeed (Primary Key, Clustered: Id)
- Check Constraints (0)
- ▲ Indexes (1)  
IX\_RssFeed\_ResourceRootId (Unique: ResourceRootId)
- ▲ Foreign Keys (1)  
FK\_RssFeed\_ResourceRoot\_ResourceRootId (Id)
- Triggers (0)

RssFeed table

dbo.ResourceRoot [Design]			
Update   Script File: dbo.ResourceRoot.sql			
	Name	Data Type	All
1	Id	int	<input type="checkbox"/>
2	Name	nvarchar(MAX)	<input type="checkbox"/>
3	Url	nvarchar(MAX)	<input type="checkbox"/>
4			<input type="checkbox"/>

- ▲ **Keys** (1)  
PK\_ResourceRoot (Primary Key, Clustered: Id)
- Check Constraints (0)
- Indexes (0)
- Foreign Keys (0)
- Triggers (0)

ResourceRoot Table



1-to-1 Relationship: ResourceRoot.Id points to RssFeed.ResourceRootId

## One to Many (Example 1)

Next, let's take a look at the One-to-Many relationship for each **ResourceCatalog** that has zero or more **ItemLists**. For example:

- Resource Catalog = ASP .NET Core Blogs
- Item List = ASP .NET Core A-Z Blog Series

In the two classes, we see the following code:

```
public class ResourceCatalog
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<ItemList> ItemLists { get; set; }
}

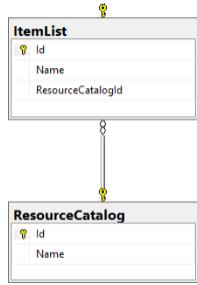
public class ItemList
```

```

{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<LearningResourceItemList> LearningResourceItemLists
    {
        get; set;
    }
}

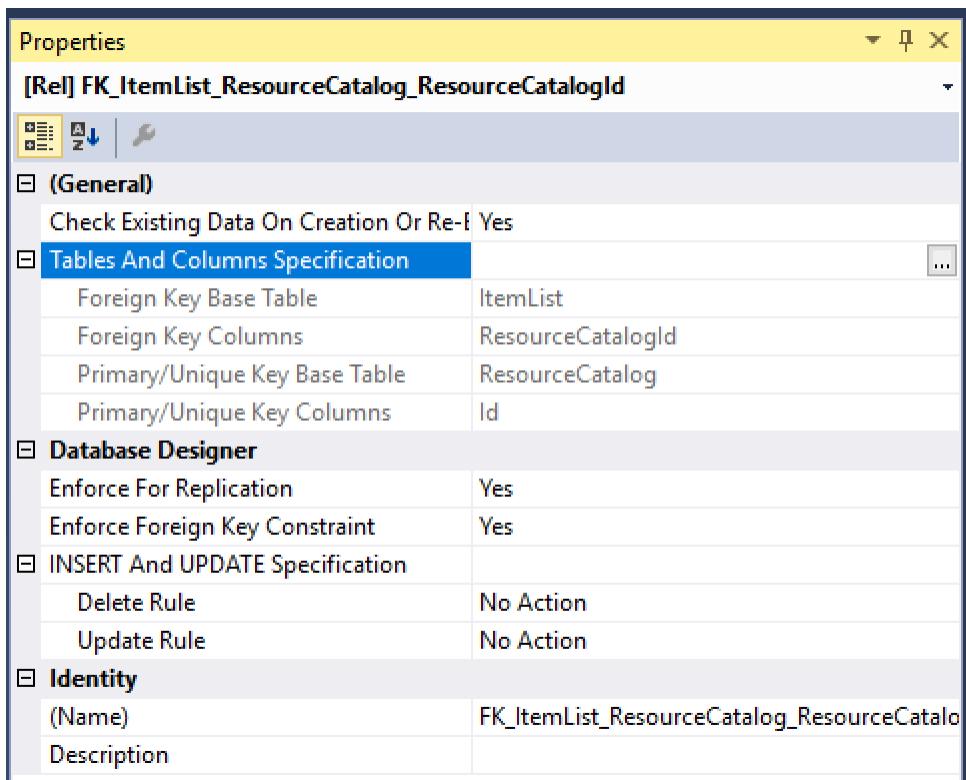
```

Each Resource Catalog has zero or more Item Lists, so the **ResourceCatalog** class has a `List<T>` property for **ItemLists**. This is even simpler than the previously described 1-to-1 relationship. In the **ItemList** class, you don't need a property pointing back to the **ResourceCatalog**.



One-to-Many Relationship

Another way of looking at the *One-to-Many* relationship is to view the constraints of each database entity in the visuals below. Note that both tables have an **Id** field that is a Primary Key (once again, inferred by EF Core) while the **ItemList** table also has a Foreign Key for the **ResourceCatalogId** field used for the constraint in the relationship.



1-to-Many Relationship: ResourceCatalog.Id points to ItemList.ResourceCatalogId

## One to Many (Example 2)

Let's also take a look at another One-to-Many relationship, for each **ResourceRoot** that has zero or more **LearningResources**. For example:

- Resource Root = Wake Up and Code! blog site
- Learning Resource = Specific blog post on site

In the two classes, we see the following code:

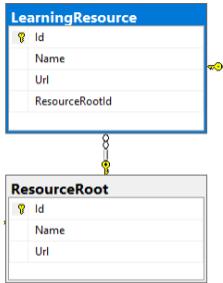
```
public class ResourceRoot: InternetResource
{
    public RssFeed RssFeed { get; set; }
    public List<LearningResource> LearningResources { get; set; }
} public class LearningResource : InternetResource
```

```

{
    public List<LearningResourceItemList> LearningResourceItemLists
    {
        get; set;
    }
}

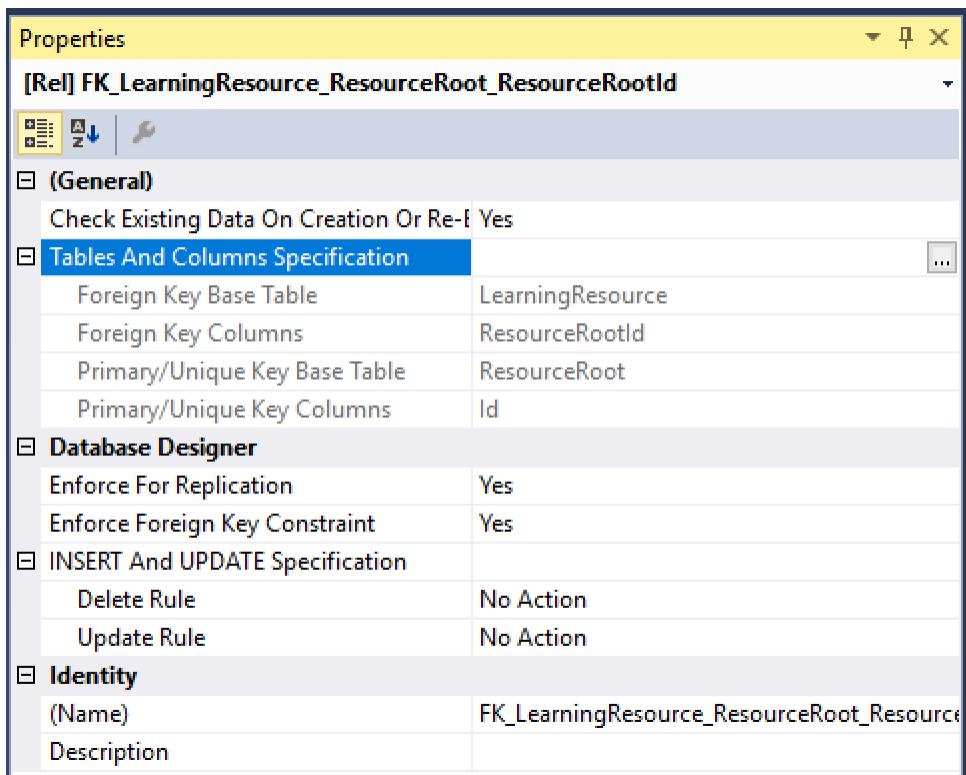
```

Each Resource Root has zero or more Learning Resources, so the **ResourceRoot** class has a `List<T>` property for **LearningResources**. This is just as simple as the aforementioned 1-to-many relationship. In the **LearningResource** class, you don't need a property pointing back to the **ResourceRoot**.



ResourceRoot and LearningResource

Another way of looking at the *One-to-Many* relationship is to view the constraints of each database entity in the visuals below. Note that both tables have an `Id` field that is a Primary Key (inferred by EF Core, as you should know by now) while the **LearningResource** table also has a Foreign Key for the **ResourceRootId** field used for the constraint in the relationship.



1-to-Many Constraint for ResourceRoot and LearningResource

## Many to Many

Finally, let's also take a look at a Many-to-Many relationship, for each **ItemList** and **LearningResource**, either of which can have many of the other. For example:

- Item List = ASP .NET Core A-Z Blog Series
- Learning Resource = Specific blog post on site

This relationship is a little more complicated than all of the above, as we will need a “join table” to connect the two tables in question. Not only that, we will have to describe the entity in the C# code with connections to both tables we would like to connect with this relationship.

If you're wondering when EF Core will support this type of relationship *without* a join table, check out the following GitHub issue discussions:

- Many-to-many relationships without an entity class to represent the join table: <https://github.com/aspnet/EntityFrameworkCore/issues/13009>
- Implement many-to-many relationships without mapping join table: <https://github.com/aspnet/EntityFrameworkCore/issues/10508>

Specifically, take a look at this comment: “*Current plan for 3.0 is to implement skip-level navigation properties as a stretch goal. If property bags (#9914) also make it into 3.0, enabling a seamless experience for many-to-many could become easier.*”

In the two classes we would like to connect, we see the following code:

```
public class ItemList
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<LearningResourceItemList> LearningResourceItemLists
    {
        get; set;
    }
} public class LearningResource : InternetResource
{
    public List<LearningResourceItemList> LearningResourceItemLists
    {
        get; set;
    }
}
```

Next, we have the `LearningResourceItemList` class as a “join entity” to connect the above:

```
public class LearningResourceItemList
{
    public int LearningResourceId { get; set; }
    public LearningResource LearningResource { get; set; }
    public int ItemListId { get; set; }
    public ItemList ItemList { get; set; }
}
```

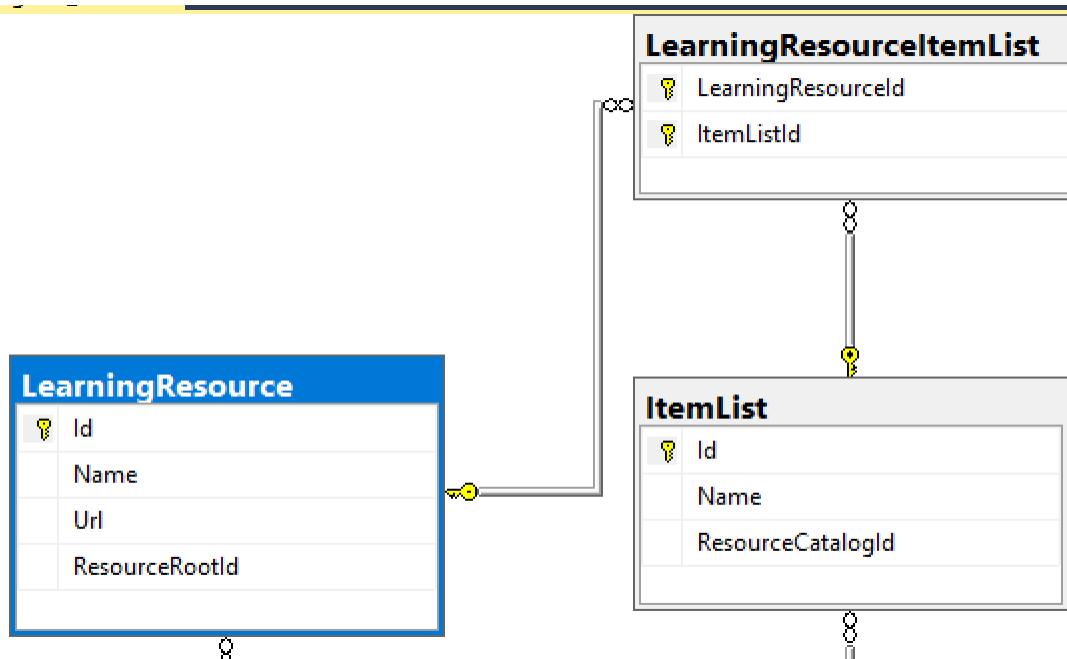
This special class has the following properties:

- **LearningResourceId**: integer value, pointing back to `LearningResource.Id`

- **LearningResource**: optional “navigation” property, reference back to connected LearningResource entity
- **ItemListId**: integer value, pointing back to ItemList.Id
- **ItemList**: optional “navigation” property, reference back to connected **ItemList** entity

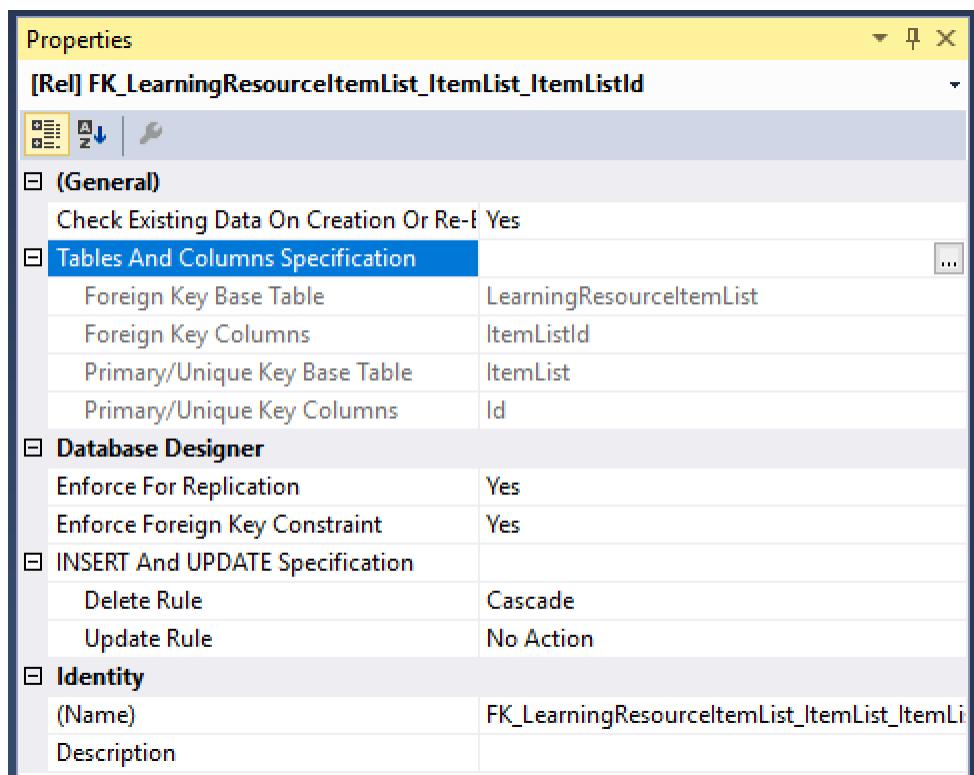
To learn more about navigation properties, check out the official docs at:

- Relationships – EF Core: <https://docs.microsoft.com/en-us/ef/core/modeling/relationships>

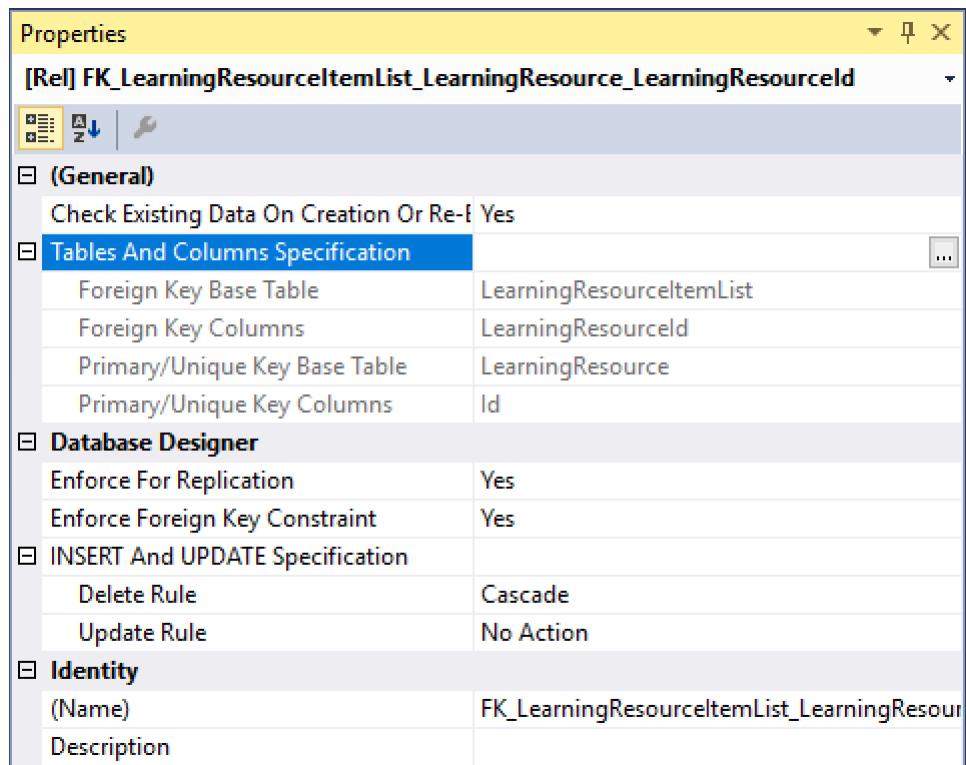


Many-to-Many Relationship

Another way of looking at the *Many-to-Many* relationship is to view the constraints of each database entity in the visuals below. Note that the two connected tables both have an **Id** field that is a Primary Key (yes, inferred by EF Core!) while the **LearningResourcelItemList** table has a *Composite Key* for the **ItemListId** and **LearningResourceId** fields used for the constraints in the relationship.



ItemList constraints



LearningResource constraints

The composite key is described in the **ApplicationDbContext** class inside the **OnModelCreating()** method:

```
public class ApplicationDbContext : IdentityDbContext
{
    ...
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<LearningResourceItemList>()
            .HasKey(r => new { r.LearningResourceId, r.ItemListId });
        base.OnModelCreating(modelBuilder);
    }
}
```

Here, the **HasKey()** method informs EF Core that the entity **LearningResourcelItemList** has a composite key defined by both **LearningResourceId** and **ItemListId**.

# References

For more information, check out the list of references below.

- Relationships – EF Core: <https://docs.microsoft.com/en-us/ef/core/modeling/relationships>
- Keys – EF Core: <https://docs.microsoft.com/en-us/ef/core/modeling/keys>
- Introduction to Relationships: <https://www.learnentityframeworkcore.com/relationships>
- Julie Lerman on Pluralsight: <https://app.pluralsight.com/profile/author/julie-lerman>
- 2.0 Getting Started: <https://app.pluralsight.com/library/courses/entity-framework-core-2-getting-started>
- 2.0 Mappings: <https://app.pluralsight.com/library/courses/e-f-core-2-beyond-the-basics-mappings>
- 2.1 What's New: <https://app.pluralsight.com/library/courses/playbook-e-f-core-2-1-whats-new>

For detailed tutorials that include both Razor Pages and MVC, check out the official tutorials below:

- New database – EF Core: <https://docs.microsoft.com/en-us/ef/core/get-started/aspnetcore/new-db?tabs=visual-studio>
- Existing Database – EF Core: <https://docs.microsoft.com/en-us/ef/core/get-started/aspnetcore/existing-db>
- ASP.NET Core MVC with EF Core: <https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc>
- ASP.NET Core Razor Pages with EF Core: <https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp>

# Forms and Fields in ASP .NET Core

By Shahed C on February 13, 2019

6 Replies

This is the **sixth** of a series of posts on ASP .NET Core for 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

- F is for Forms (and Fields)
- Tag Helpers for HTML form elements
- Input Tag Helper
- Checkboxes
- Hidden Fields
- Radio Buttons
- Textarea Tag Helper
- Label Tag Helper
- Select Tag Helper
- MVC Sample

- Razor Pages with BindProperty
- References

This article will refer to the following sample code on GitHub:



Forms And Bindings: <https://github.com/shahedc/FormsAndBindings>

# F is for Forms (and Fields)

Before Tag Helpers were available, you would have to use HTML Helper methods to create forms and their elements in a ASP .NET Core views. This meant that your form could look something like this:

```
@using (Html.BeginForm())
{
    <input />
}
```

With the introduction of Tag Helpers, you can now make your web pages much more cleaner. In fact, Tag Helpers work with both MVC Views and Razor Pages. The syntax is much simpler:

```
<form method="post">
```

This looks like HTML because it is HTML. You can add additional server-side attributes within the `<form>` tag for additional features.

```
<form asp-controller="ControllerName" asp-action="ActionMethod"
method="post">
```

In this above example, you can see how the attributes **asp-controller** and **asp-action** can be used to specify a specific controller name and action method. When these optional attributes are omitted, the current controller and default action method will be used.

Optionally, you can also use a **named route**, e.g.

```
<form asp-route="NamedRoute" method="post">
```

The `asp-route` attribute will look for a specific route with the name specified. When the form is submitted via HTTP POST, the action method will then attempt to read the form values via a passed values or bound properties.

In a Controller's class file within an MVC app, you can set an optional **Name** for your action method's **Route** attribute, as shown below:

```
[Route("/ControllerName/ActionMethod", Name = "NamedRoute")]
public IActionResult ActionMethod()
{}
```

While you won't find new Tag Helper equivalents for each and every HTML Helper you may have used in the past, you should consider using a Tag Helper wherever possible. You can even create your own custom Tag Helpers as well. For more information on custom Tag Helpers, check out the official documentation:

- Author Tag Helpers: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/authoring>

# Tag Helpers for HTML form elements

Below is a list of Tag Helpers with their corresponding HTML form elements:

## Input Tag Helper

Let's say you have a model with a couple of fields:

```
public class MyModel
{
    public string MyProperty1 { get; set; }
    public string MyProperty2 { get; set; }
}
```

You can use the following syntax to use an Input Tag Helper with an expression name assigned to the **asp-for** attribute. This allows you to refer to the properties without requiring the “Model.” prefix in your Views and Pages.

```
@model MyModel
...
<!-- Syntax -->
<input asp-for="<Expression Name>" />
...
<!-- Examples -->
<input asp-for="MyProperty1" />
<input asp-for="MyProperty2" />
```

Corresponding to the Input Tag Helper, there are existing HTML Helpers, with some differences:

- `Html.TextBox`: doesn't automatically set the type attribute
- `Html.TextBoxFor`: also doesn't automatically set the type attribute; strongly typed

- `Html.Editor`: suitable for collections, complex objects and templates (while `Input Tag Helper` is not).
- `Html.EditorFor`: also suitable for collections, complex objects and templates; strongly typed

Since `Input Tag Helpers` use an inline variable or expression in your `.cshtml` files, you can assign the value using the `@` syntax as shown below:

```
@{
    var myValue = "Some Value";
}
<input asp-for="@myValue" />
```

This will generate the following textbox input field:

```
<input type="text" id="myValue" name="myValue" value="Some Value" />
```

To create more specific fields for email addresses, passwords, etc, you may use `data-type` attributes on your models to auto-generate the necessary fields. These may include one of the following enum values:

- `CreditCard`
- `Currency`
- `Custom`
- `Date`
- `DateTime`
- `Duration`
- `EmailAddress`
- `Html`
- `ImageUrl`
- `MultilineText`
- `Password`
- `PhoneNumber`
- `PostalCode`
- `Text`
- `Time`
- `Upload`

- Url

```
// For example:  
[DataType(DataType.Date)]  
public DateTime DateOfBirth { get; set; }
```

Note that each attribute can be applied on a field for the view/page generator to infer the data type, but is **not** used for data validation. For validation, you should use the appropriate validation techniques in your code. We will cover validation in a future blog post, but you can refer to the official docs for now:

- Model validation in ASP.NET Core MVC: <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation>
- Add validation to an ASP.NET Core Razor Page: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/validation>

## Checkboxes

Any boolean field in your model will automatically be turned into a checkbox in the HTML form. There is no extra work necessary to specify that the input type is a “checkbox”. In fact, the generated HTML includes the “checkbox” type automatically, sets the “checked” property if checked and wraps it in a label with the appropriate caption. For example, imagine a boolean field named “IsActive”:

```
// boolean field in a model class  
public bool IsActive { get; set; } <!-- input field in page/view  
wrapped in label -->  
<label class="form-check-label">  
  <input class="form-check-input" asp-for="IsActive" />  
  @Html.DisplayNameFor(model => model.IsActive)  
</label> <!-- HTML generated for boolean field -->  
<label class="form-check-label">  
  <input  
    class="form-check-input"  
    type="checkbox"  
    checked="checked"  
    data-val="true"  
    data-val-required="The IsActive field is required."  
    id="IsActive"  
    name="IsActive"  
    value="true"> IsActive  
</label>
```

## Hidden Fields

In case you're wondering how you can generate a *hidden* `<input>` field, you can simply use the `[HiddenInput]` attribute on your hidden field's property, as shown below. If you wish, you can explicitly set “`type=hidden`” in your Page/View, but I prefer to set the attribute in the model itself.

```
// hidden property in model class
[HiddenInput]
public string SomeHiddenField { get; set; } = "Some Value"; <!-- hidden
field in page/view -->
<input asp-for="SomeHiddenField" /> <!-- HTML generated for hidden
field -->
<input type="hidden" id="SomeHiddenField" name="SomeHiddenField"
value="Some Value">
```

## Radio Buttons

For radio buttons, you can create one `<input>` tag for each radio button option, with a reference to a common field, and a unique value for each radio button. Each input element can be wrapped in a label to include a proper (clickable) text caption . You can generate these in a loop or from a collection from dynamically generated radio buttons.

```
// string property in model class
public string ExperienceLevel { get; set; } <!-- input fields for radio
buttons in page/view -->
<label><input asp-for="ExperienceLevel" value="N" type="radio"
/>Novice</label>
<label><input asp-for="ExperienceLevel" value="B"
type="radio"/>Beginner</label>
<label><input asp-for="ExperienceLevel" value="I" type="radio"
/>Intermediate</label>
<label><input asp-for="ExperienceLevel" value="A"
type="radio"/>Advanced</label> <!-- HTML generated for radio buttons -->
<label><input value="N" type="radio" id="ExperienceLevel"
name="ExperienceLevel">Novice</label>
<label><input value="B" type="radio" id="ExperienceLevel"
name="ExperienceLevel">Beginner</label>
<label><input value="I" type="radio" id="ExperienceLevel"
name="ExperienceLevel">Intermediate</label>
<label><input value="A" type="radio" id="ExperienceLevel"
name="ExperienceLevel">Advanced</label>
```

## Textarea Tag Helper

The multiline <textarea> field can be easily represented by a **Textarea Tag Helper**. This is useful for longer strings of text that need to be seen and edited across multiple lines.

```
public class MyModel
{
    [MinLength(5)]
    [MaxLength(1024)]
    public string MyLongTextProperty { get; set; }
}
```

As you would expect, you can use the following syntax to use a Textarea Tag Helper with an expression name assigned to the **asp-for** attribute.

```
@model MyModel
...
<textarea asp-for="MyLongTextProperty"></textarea>
```

This will generate the following textarea input field:

```
<textarea
    data-val="true"
    data-val-maxlength="The field ... maximum length of '1024'."
    data-val-maxlength-max="1024"
    data-val-minlength="The field ... minimum length of '5'."
    data-val-minlength-min="5"
    id="MyLongTextProperty"
    maxlength="1024"
    name="MyLongTextProperty"
></textarea>
```

Note that the property name and its attributes are used to create that textarea with the necessary id, name, maxlength and data validation settings.

Corresponding to the Textarea Tag Helper, the existing HTML Helper is shown below:

- `Html.TextAreaFor`

## Label Tag Helper

The `<label>` field can be represented by a **Label** Tag Helper. A label usually goes hand-in-hand with a specific `<input>` field, and is essential in creating text captions for more accessible web applications. The **Display** attribute from your model's fields are used for the label's displayed text values. (*You could use the **DisplayName** attribute instead and omit the **Name** parameter, but it limits your ability to use localized resources.*)

```
public class MyModel
{
    [Display(Name = "Long Text")]
    public string MyLongTextProperty { get; set; }
}
```

You can use the following syntax to use a **Label** Tag Helper along with an **Input** Tag Helper.

```
@model MyModel
...
<label asp-for="MyLongTextProperty"></label>
<input asp-for="MyLongTextProperty" />
```

This will generate the following HTML elements:

```
<label for="MyLongTextProperty">Long Text</label>
<input type="text" id="MyLongTextProperty" name="MyLongTextProperty"
value="">
```

Note that the property name and its attributes are used to create both the label with its descriptive caption and also the input textbox with the necessary id and name.

Corresponding to the Label Tag Helper, the existing HTML Helper is shown below:

- `Html.LabelFor`

## Select Tag Helper

The `<select>` field (with its nested `<option>` fields) can be represented by a Select Tag Helper. This visually represents a dropdown or listbox, from which the user may select one or more options. In your model, you can represent this with a `List<SelectListItem>` of items, made possible by the namespace `Microsoft.AspNetCore.Mvc.Rendering`.

```
...
using Microsoft.AspNetCore.Mvc.Rendering;
```

```

public class MyModel
{
    public string MyItem { get; set; }

    public List<SelectListItem> MyItems { get; } = new
List<SelectListItem>
{
    new SelectListItem { Value = "Item1", Text = "Item One" },
    new SelectListItem { Value = "Item2", Text = "Item Two" },
    new SelectListItem { Value = "Item3", Text = "Item Three" },
};

}

```

You can use the following syntax to use a Select Tag Helper.

```

@model MyModel
...
<select asp-for="MyItem" asp-items="Model.MyItems"></select>

```

Note that the **asp-items** attribute **does** require a “Model.” prefix, unlike the **asp-for** attribute that we have been using so far. This will generate the following HTML:

```

<select id="MyItem" name="MyItem">
    <option value="Item1">Item One</option>
    <option value="Item2">Item Two</option>
    <option value="Item3">Item Three</option>
</select>

```

Note that the property name and its attributes are used to create both the dropdown list and also the nested options available for selection. For more customization, optgroups and multiple selections, check out the “Select Tag Helper” section in the Tag Helpers documentation at:

- [Select Tag Helper section] Tag Helpers in forms: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2#the-select-tag-helper>

Corresponding to the Select Tag Helper, the existing HTML Helpers are shown below:

- `Html.DropDownListFor`
- `Html.ListBoxFor`

# MVC Sample

In the sample repo, you'll find an MVC web project with various models, views and controllers.

- **Models:** In the “Models” folder, you’ll find a Human.cs class (shown below) with some fields we will use to display HTML form elements.
- **Views:** Within the “Views” subfolder, the “Human” subfolder contains auto-generated views for the HumanController’s methods, while the “Attr” subfolder contains simple Index and Detail views to be used by the AttrController class.
- **Controllers:** The HumanController class was auto-generated for the Human model, while the AttrController class was manually written to illustrate the use of various attributes (e.g. FromQuery, FromRoute, FromForm) within the action methods.

```
public class Human
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public DateTime DateOfBirth { get; set; }
    public int FavoriteNumber { get; set; }
    public bool IsActive { get; set; }
}
```

Take a look at the Create and Edit views for the Human class, and you’ll recognize familiar sets of **<label>** and **<input>** fields that we discussed earlier.

```
...
<div class="form-group">
    <label asp-for="FirstName" class="control-label"></label>
    <input asp-for="FirstName" class="form-control" />
    <span asp-validation-for="FirstName" class="text-danger"></span>
</div>
...
```

To bind these fields, the **Create** and **Edit** methods that respond to HTTP POST both use a **[Bind]** attribute for the **Human** parameter with specific fields to bind from the model:

```
[HttpPost]
public async Task<IActionResult> Create(
    [Bind("Id,FirstName,LastName,Address,DateOfBirth,FavoriteNumber,IsActive")]
    Human human)
```

```
...
[HttpPost]
public async Task<IActionResult> Edit(
    int id,
    [Bind("Id,FirstName,LastName,Address,DateOfBirth,FavoriteNumber,IsActive")]
    Human human)
```

In the AttrController class, you'll find a different approach to gathering information submitted in an HTML form.

- **FromQuery:** The **Index()** method uses a **[FromQuery]** attribute which can obtain information from the URL's QueryString parameters
- **FromRoute:** The first **Details()** method uses a **[FromRoute]** attribute which can obtain information from route values.
- **FromForm:** The second **Details()** method uses a **[FromForm]** attribute which links the submitted form fields to the fields of the corresponding model passed to the method.

```
[HttpGet]
public IActionResult Index([FromQuery] string humanInfo) { }
...
[HttpGet]
public IActionResult Details([FromRoute] int id) { }
...
[HttpPost]
public IActionResult Details([FromForm] Human human) { }
...
```

The first two methods above can be reached via HTTP GET requests by accessing a URL, while the third one can be reached when submitting the Details form with a submit button. Run the sample application to click around and view its behavior.

Details - FormsAndBindings

https://localhost:44316/Attr/Details/1

FirstName  
John

LastName  
Doe

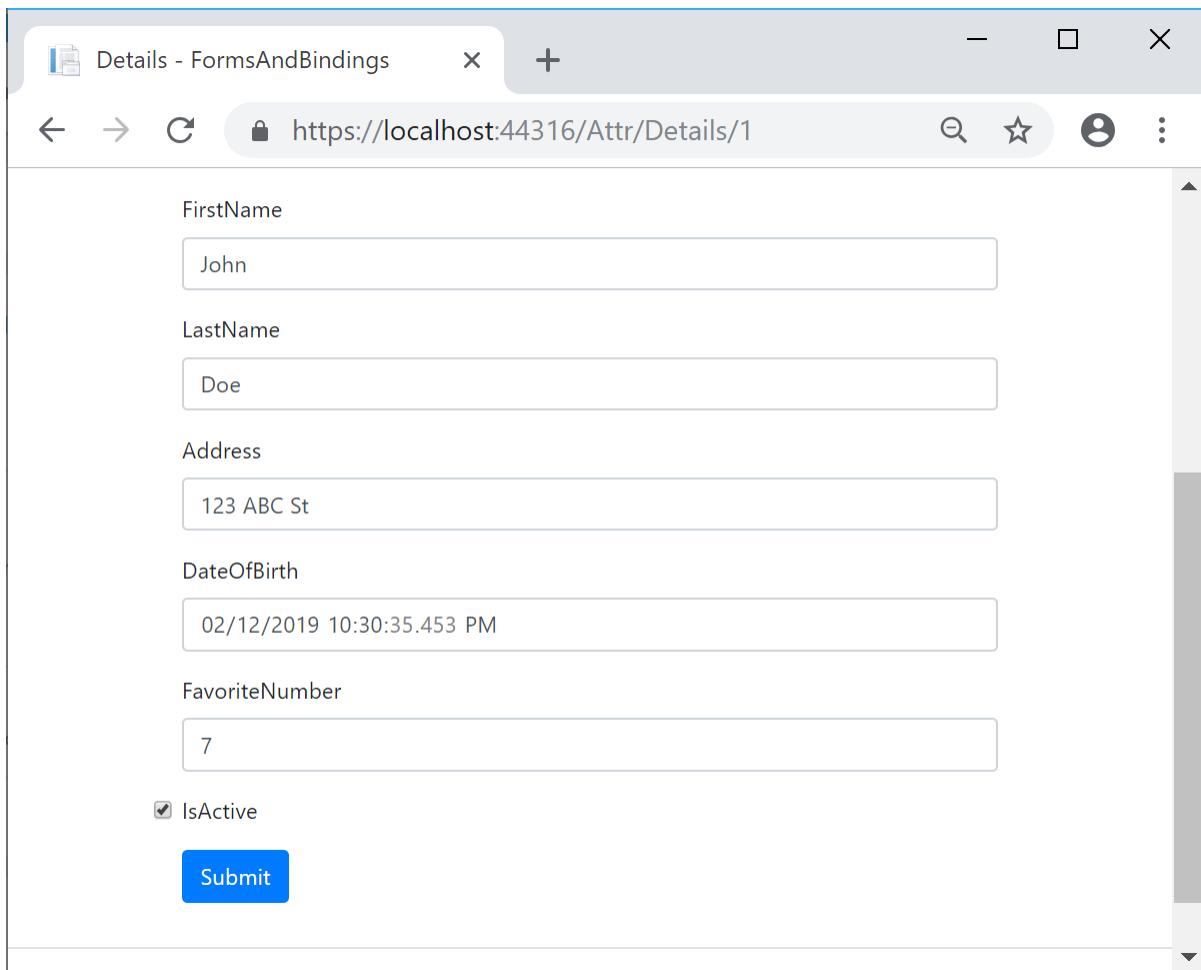
Address  
123 ABC St

DateOfBirth  
02/12/2019 10:30:35.453 PM

FavoriteNumber  
7

IsActive

Submit



```
<!-- From shared _Layout.cshtml, link to Index page -->
<a
    class="nav-link text-dark"
    asp-area=""
    asp-controller="Attr"
    asp-action="Index"
    asp-route-humanInfo="John"
>Attributes</a>

<!-- From Index.cshtml, link to Details page -->
<a
    class="nav-link text-dark"
    asp-area=""
    asp-controller="Attr"
    asp-action="Details"
    asp-route-id="1"
>View Details</a><!-- From Details.cshtml, self-submitting form -->
```

```
<form asp-action="Details">
...
<input type="submit" value="Submit" class="btn btn-primary" />
...
</form>
```

Note that `<a>` tags have an `asp-route-xx` attribute that can have any text value appended to the end of it. These route parameters, e.g. `id` and `humanInfo`, correspond directly to action method parameters seen in `Index()` and `Details()` methods of the `AttrController` class.

## Razor Pages with `BindProperty`

Compared to MVC views, the newer Razor Pages make it a lot easier to bind your model properties to your HTML forms. The `[BindProperty]` attribute can be applied to MVC Controllers as well, but is much more effective within Razor Pages.

In the sample repo, you'll find a Razor web project with multiple subfolders, including Models and Pages.

- Models: In the “Models” folder of the Razor web project, you’ll find a `Human.cs` class (shown below) with some fields we will use to display HTML form elements.
- Pages: Within the “Pages” subfolder, the “Human” subfolder contains auto-generated Razor Pages along with corresponding .cs classes that contain the necessary Get/Post methods.

Here, the `HumanModel` class is named slightly differently from the aforementioned (MVC) example’s `Human` class, since the subfolder within Pages is also called `Human`. (*We would have to rename one or the other to avoid a naming conflict.*)

```
public class HumanModel
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public DateTime DateOfBirth { get; set; }
    public int FavoriteNumber { get; set; }
    public bool IsActive { get; set; }
}
```

This time, take a look at the Create and Edit pages for the Human class, and you'll once again recognize familiar sets of `<label>` and `<input>` fields that we discussed earlier.

```
<div class="form-group">
    <label asp-for="HumanModel.FirstName" class="control-label"></label>
    <input asp-for="HumanModel.FirstName" class="form-control" />
    <span asp-validation-for="HumanModel.FirstName" class="text-danger"></span>
</div>
```

Since there are no controller classes in the Razor web project, let's take a look at the corresponding C# classes for the **Create** and **Edit** pages, i.e. Create.cshtml.cs and Edit.cshtml.cs. In both of these classes, we'll find the **[BindProperty]** attribute in use, right after the constructor and before the Get/Post methods.

```
[BindProperty]
public HumanModel HumanModel { get; set; }
```

This **[BindProperty]** attribute allows you to declaratively bind the HumanModel class and its properties for use by the HTML form in the corresponding Razor Page. This is an opt-in feature that allows to choose which properties to bind. If you wish, you could alternatively bind *all* public properties in the class by using the **[BindProperties]** attribute above the class, instead of above each individual member.

**NOTE:** By default, a Razor Page's default methods for HTTP GET and HTTP POST are `OnGet()` and `OnPost()` respectively. If you wish to use custom page handlers in your HTML forms, you must create custom methods with the prefix `OnPost` followed by the name of the handler (and optionally followed by the word `Async` for `async` methods)

```
<!-- buttons with custom page handlers -->
<input type="submit" asp-page-handler="Custom1" value="Submit 1" />
<input type="submit" asp-page-handler="Custom2" value="Submit 2" /> // action methods in .cs file associated with a Razor Page
public async Task<IActionResult> OnPostCustom1Async() { }
public async Task<IActionResult> OnPostCustom2sync() { }
```

The standard set of Get/Post methods are shown below, from Create.cshtml.cs:

```
public IActionResult OnGet()
{
    return Page();
} public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }
    _context.HumanModel.Add(HumanModel);
```

```
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
```

Note that the **HumanModel** is passed to the DB Context to add it to the database. If you were to remove the aforementioned **[BindProperty]** attribute, HumanModel would be null and the save operation would fail. The above approach only opts in to accepting HTTP POST requests. To enable use of **BindProperty** for HTTP GET requests as well, simply set the optional parameter **SupportsGet** to true, as shown below.

```
[BindProperty(SupportsGet = true)]
```

## References

- Tag Helpers in forms: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms>
- Anchor Tag Helper: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/anchor-tag-helper>
- Model Binding in MVC: <https://exceptionnotfound.net/asp-net-core-demystified-model-binding-in-mvc/>
- Model Binding in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding>
- BindProperty for Controllers or Razor Pages: <http://www.binaryintellect.net/articles/85fb9a1d-6b0d-4d1f-932c-555bd27ba401.aspx>
- Model Binding in Razor Pages: <https://www.learnrazorpageds.com/razor-pages/model-binding>
- Introduction to Razor Pages: <https://docs.microsoft.com/en-us/aspnet/core/razor-pages>
- The ASP.NET Core Form Tag Helpers Cheat Sheet: <https://jonthilton.net/aspnet-core-forms-cheat-sheet/>

# Generic Host Builder in ASP .NET Core

By Shahed C on February 18, 2019

5 Replies

This is the **seventh** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**

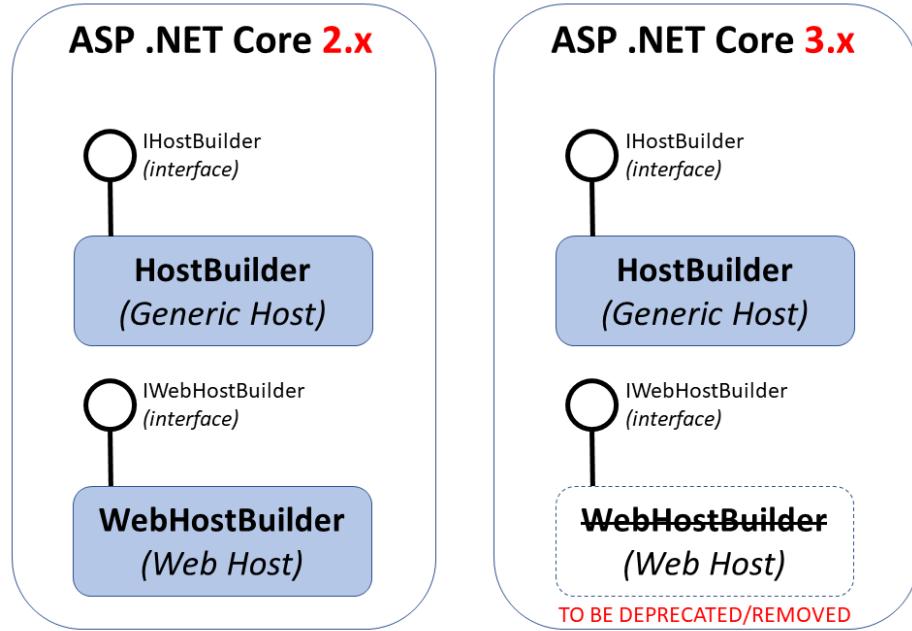


## In this Article:

- G is for Generic Host Builder
- Generic Host Builder in 2.x
- Web Host Builder in 2.x
- Generic Host Builder for Web Apps in 3.x
- References

## G is for Generic Host Builder

The Generic Host Builder in ASP .NET Core was introduced in v2.1, but only meant for non-HTTP workloads. However, it is intended to replace the Web Host Builder when v3.0 is released in 2019.



Generic Host Builder in ASP .NET Core 3.0

## Generic Host Builder in 2.x

So, if the Generic Host Builder isn't currently used for web hosting in v2.x, what can it be used for? The aforementioned non-HTTP workloads include a number of capabilities according to the documentation, including:

- app config, e.g. set base path, add hostsettings.json, env variables, etc
- dependency injection, e.g. various hosted services
- logging capabilities, e.g. console logging

The **HostBuilder** class is available from the following namespace, implementing the **IHostBuilder** interface:

```
using Microsoft.Extensions.Hosting;
```

At a minimum, the **Main()** method of your .NET Core app would look like the following:

```
public static async Task Main(string[] args)
{
    var host = new HostBuilder()
        .Build();

    await host.RunAsync();
}
```

Here, the **Build()** method initializes the host, so (as you may expect) it can only be called once for initialization. Additional options can be configured by calling the **ConfigureServices()** method before initializing the host with **Build()**.

```
var host = new HostBuilder()
    .ConfigureServices((hostContext, services) =>
{
    services.Configure<HostOptions>(option =>
    {
        // option.SomeProperty = ...
    });
})
.Build();
```

Here, the **ConfigureServices()** method takes in a **HostBuilderContext** and an injected collection of **IServiceCollection** services. The options set in the **Configure()** can be used to set additional **HostOptions**. Currently, **HostOptions** just has one property, i.e. **ShutdownTimeout**.

You can see more configuration capabilities in the official sample, broken down into the snippets below:

#### **Host Config Snippet:**

```
.ConfigureHostConfiguration(configHost =>
{
    configHost.SetBasePath(Directory.GetCurrentDirectory());
    configHost.AddJsonFile("hostsettings.json", optional: true);
    configHost.AddEnvironmentVariables(prefix: "PREFIX_");
    configHost.AddCommandLine(args);
})
```

#### **App Config Snippet:**

```

.ConfigureAppConfiguration((hostContext, configApp) =>
{
    configApp.AddJsonFile("appsettings.json", optional: true);
    configApp.AddJsonFile(
        $"appsettings.{hostContext.HostingEnvironment.EnvironmentName}.json",
        optional: true);
    configApp.AddEnvironmentVariables(prefix: "PREFIX_");
    configApp.AddCommandLine(args);
})

```

**Dependency Injection Snippet:**

```

.ConfigureServices((hostContext, services) =>
{
    services.AddHostedService<LifetimeEventsHostedService>();
    services.AddHostedService<TimedHostedService>();
})

```

**Logging Snippet:**

```

.ConfigureLogging((hostContext, configLogging) =>
{
    configLogging.AddConsole();
    configLogging.AddDebug();
})

```

## Web Host Builder in 2.x

The **WebHostBuilder** class is available from the following namespace (specific to ASP .NET Core), implementing the **IWebHostBuilder** interface:

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

The Web Host Builder in ASP .NET Core is currently used for hosting web apps as of v2.x. As mentioned in the previous section, it will be replaced by the Generic Host Builder in v3.0. At a minimum, the **Main()** method of your ASP .NET Core 2.x web app would look like the following:

```

public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)

```

```
=>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
}
```

If you're not familiar with the shorthand syntax of the helper method **CreateWebHostBuilder()** shown above, here's what it would normally look like, expanded:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args)
{
    return WebHost.CreateDefaultBuilder(args).UseStartup<Startup>();
}
```

**NOTE:** This type of C# syntax is known as an Expression Body Definition, introduced for methods in C# 6.0, and additional features in C# 7.0.

The **CreateDefaultBuilder()** method performs a lot of “magic” behind the scenes, by making use of pre-configured defaults. From the official documentation, here is a summary of the default configuration from the Default Builder:

- use Kestrel as the web server
- configure it using the application’s configuration providers,
- set the ContentRootPath to the result of GetCurrentDirectory(),
- load IConfiguration from ‘appsettings.json’ and ‘appsettings.[EnvironmentName].json’,
- load IConfiguration from User Secrets when EnvironmentName is ‘Development’ using the entry assembly,
- load IConfiguration from environment variables,
- load IConfiguration from supplied command line args,
- configure the ILoggerFactory to log to the console and debug output,
- enable IIS integration (if running behind IIS with the ASP .NET Core Module)

For more information on some of the above, here are some other blog posts in this series, you may find useful:

- Your Web App Secrets in ASP .NET Core
- (Coming soon) IIS In-Process Modules in ASP .NET Core

# Generic Host Builder for Web Apps in 3.x

Going forward, ASP .NET Core 3.0 will allow you to use the updated Generic Host Builder instead of the Web Host Builder in your web apps. As of Preview 2, the templates available in ASP .NET Core 3.0 have already been updated to include the Generic Host Builder.

At a minimum, the **Main()** method of your .NET Core 3.0 web app would now look like the following:

```
public static void Main(string[] args)
{
    CreateHostBuilder(args)
        .Build()
        .Run();
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
}
```

Here's an expanded representation of the **CreateHostBuilder()** method:

```
public static IHostBuilder CreateHostBuilder(string[] args)
{
    return Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
}
```

This **CreateHostBuilder()** method in the 3.0 template looks very similar to the 2.x call to **CreateWebHostBuilder()** mentioned in the previous section. In fact, the main difference is that the call to **WebHost.CreateDefaultBuilder()** is replaced by **Host.CreateDefaultBuilder()**. Using the **CreateDefaultBuilder()** helper method makes it very easy to switch from v2.x to v3.0.

Another difference is the call to **ConfigureWebHostDefaults()**. Since the new host builder is a **Generic** Host Builder, it makes sense that we have to let it know that we intend to configure the default settings for a Web Host. The **ConfigureWebHostDefaults()** method does just that.

Going forward, it's important to know the following:

- **WebHostBuilder** will be *deprecated* and then *removed* in the near future
- However, the **IWebHostBuilder** interface will remain
- You won't be able to inject just any service into the Startup class...
- ... instead, you have **IHostingEnvironment** and **IConfiguration**

If you're wondering about the reason for the limitation for injecting services, this change prevents you from injecting services into the Startup class *before* **ConfigureServices()** gets called.

## References

- ASP.NET Core updates in .NET Core 3.0 Preview 2:  
<https://blogs.msdn.microsoft.com/webdev/2019/01/29/aspnet-core-3-preview-2/>
- .NET Generic Host: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-2.2>
- ASP.NET Core Web Host: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/web-host?view=aspnetcore-2.2>
- Using HostBuilder and the Generic Host in .NET Core Microservices: <https://www.stevejgordon.co.uk/using-generic-host-in-dotnet-core-console-based-microservices>
- The ASP.NET Core Generic Host: namespace clashes and extension methods: <https://andrewlock.net/the-asp-net-core-generic-host-namespace-clashes-and-extension-methods/>
- Samples on GitHub: <https://github.com/aspnet/Docs/tree/master/aspnetcore/fundamentals/host/generic-host/samples/>

# Handling Errors in ASP .NET Core

By Shahed C on February 25, 2019

5 Replies

This is the **eighth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

- H is for Handling Errors
- Exceptions with Try catch Finally
- Try-Catch-Finally in Sample App
- Error Handling for MVC
- Error Handling for Razor Pages
- Logging Errors
- Transient Fault Handling
- References

This article will refer to the following sample code on GitHub:



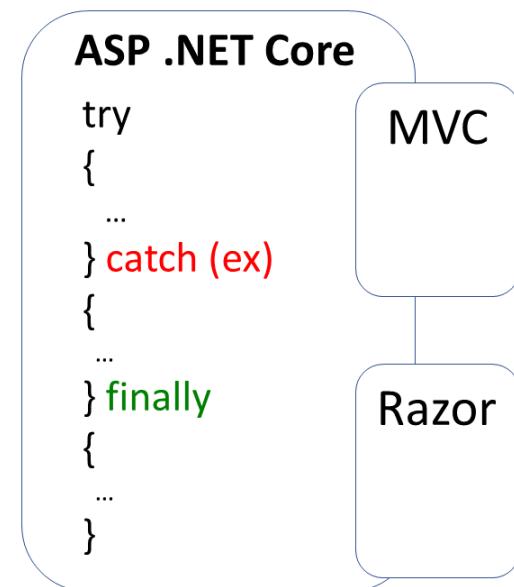
Error Handling Samples: <https://github.com/shahedc/ErrorHandlingWeb>

## H is for Handling Errors

Unless you're perfect 100% of the time (who is?), you'll most likely have errors in your code. If your code doesn't build due to compilation errors, you can probably correct that by fixing the offending code. But if your application encounters runtime errors while it's being used, you may not be able to anticipate every possible scenario.

Runtime errors may cause Exceptions, which can be caught and handled in many programming languages. Unit tests will help you write better code, minimize errors and create new features with confidence. In the meantime, there's the good ol' try-catch-finally block, which should be familiar to most developers.

**NOTE:** You may skip to the next section below if you don't need this refresher.



# Exceptions with Try-Catch-Finally

The simplest form of a try-catch block looks something like this:

```
try
{
    // try something here

} catch (Exception ex)
{
    // catch an exception here
}
```

You can chain multiple catch blocks, starting with more specific exceptions. This allows you to catch more generic exceptions toward the end of your try-catch code. In a string of **catch()** blocks, only the caught exception (if any) will cause that block of code to run.

```
try
{
    // try something here

} catch (IOException ioex)
{
    // catch specific exception, e.g. IOException

} catch (Exception ex)
{
    // catch generic exception here
}
```

Finally, you can add the optional **finally** block. Whether or not an exception has occurred, the finally block will always be executed.

```
try
{
    // try something here

} catch (IOException ioex)
{
    // catch specific exception, e.g. IOException
```

```
} catch (Exception ex)
{
    // catch generic exception here

} finally
{
    // always run this code

}
```

## Try-Catch-Finally in Sample App

In the MVC sample app, the Reader Controller uses a Data Service from a shared .NET Standard Library to read from a data file that may exist in the Web App's static files. It displays a view with some hard-coded data and tries to replace some data with additional information obtained from the data file.

```
// hard-code some data
var data1 = new DataItem
{
    Id = 1,
    SomeData = "data 1 initialized"
};
var data2 = new DataItem
{
    Id = 2,
    SomeData = "data 2 initialized"
};
var data3 = new DataItem
{
    Id = 3,
    SomeData = "data 3 initialized"
};
```

Then, try to read some data from a data file, to replace information in the data model.

```
// get data from file if possible
try
{ // Open the text file using a stream reader.
    var webRoot = _env.WebRootPath;
    var file = System.IO.Path.Combine(webRoot, @"data\datafile.txt");

    using (StreamReader sr = new StreamReader(file))
```

```

{
// Read the stream to a string, overwrite some data
data2.SomeData = sr.ReadToEnd();
}
}
catch (IOException ioException)
{
    data2.SomeData = $"IO Error: {ioException.Message}";
}
catch (Exception exception)
{
    data2.SomeData = $"Generic Error: {exception.Message}";
}
finally
{
    data3.SomeData = "All done!";
}

```

In the above code, you can see a series of try-catch blocks, ending with a finally block:

1. try to read the file and overwrite some data in the 2nd data object.
2. catch an **IOException**, capture the error message
3. catch a generic **Exception** if a different exception has occurred, capture the generic error message
4. ***finally***, overwrite some data in the 3rd data object whether or not any errors have occurred.

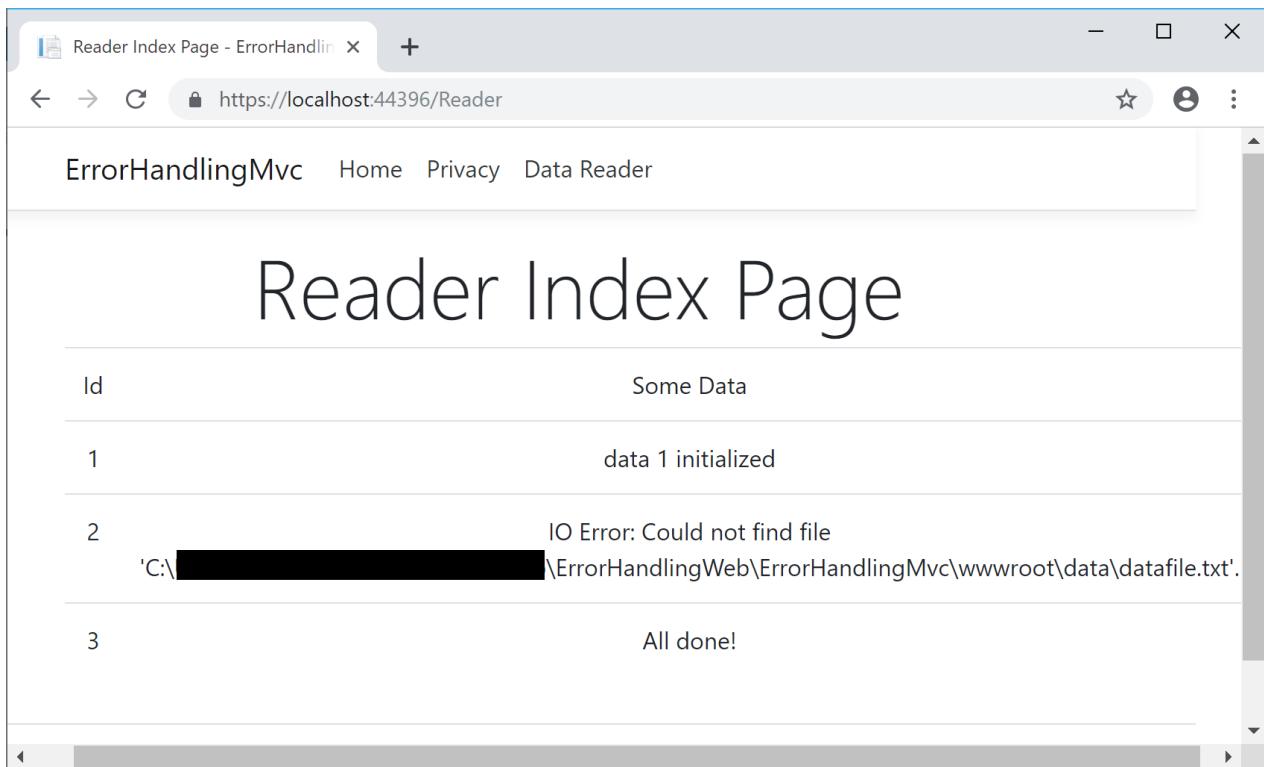
Run the MVC app and navigate to the Reader Controller. If there are no errors, you should see just the hard-coded data, with some data replaced from the data file.

The screenshot shows a web browser window titled "Reader Index Page - ErrorHandlingMvc". The address bar displays the URL "https://localhost:44396/Reader". The page content includes a navigation bar with links to "ErrorHandlingMvc", "Home", "Privacy", and "DataReader". Below the navigation bar is a large heading "Reader Index Page". A table is displayed with three rows, each containing an "Id" and a corresponding "Some Data" value. The rows are as follows:

Id	Some Data
1	data 1 initialized
2	Data from text file
3	All done!

At the bottom of the page, there is a copyright notice: "© 2019 - ErrorHandlingMvc - [Privacy](#)".

If you rename/delete the data file, then run the program again, you should see an error message as well. This reflects the IOException that occurred while attempting to read a missing file.



## Error Handling for MVC

In ASP .NET Core MVC web apps, unhandled exceptions are typically handled in different ways, depending on whatever environment the app is running in. The default template uses the **DeveloperExceptionPage** middleware in a development environment but redirects to a shared Error view in non-development scenarios. This logic is implemented in the **Configure()** method of the `Startup.cs` class.

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Home/Error");
    ...
}
```

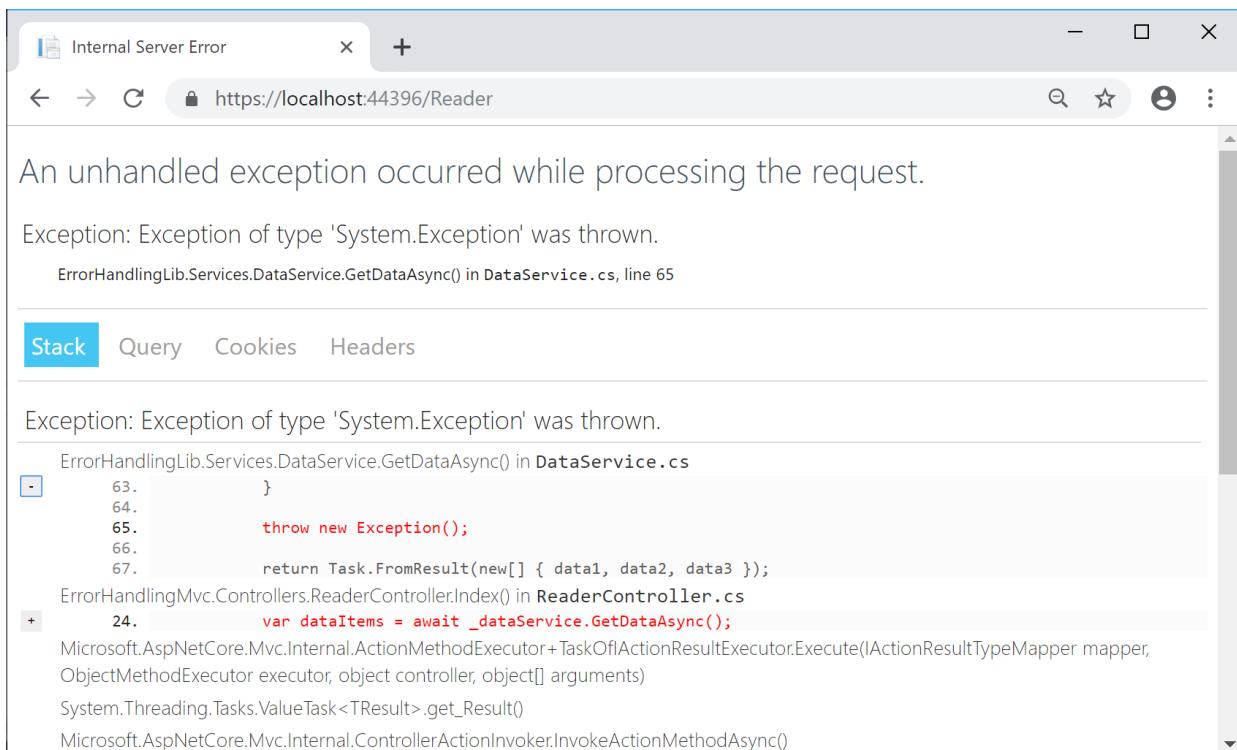
The **DeveloperExceptionPage** middleware can be further customized with **DeveloperExceptionPageOptions** properties, such as **FileProvider** and **SourceCodeLineCount**.

```

var options = new DeveloperExceptionPageOptions
{
    SourceCodeLineCount = 2
};
app.UseDeveloperExceptionPage(options);

```

Using the snippet shown above, the error page will show the offending line in red, with a variable number of lines of code above it. The number of lines is determined by the value of **SourceCodeLineCount**, which is set to 2 in this case. In this contrived example, I'm forcing the exception by throwing a new Exception in my code.



For non-dev scenarios, the shared **Error** view can be further customized by updating the **Error.cshtml** view in the **Shared** subfolder. The **ErrorViewModel** has a **ShowRequestId** boolean value that can be set to **true** to see the **RequestId** value.

```

@model ErrorViewModel
 @{
     ViewData["Title"] = "Error";
 }

<h1 class="text-danger">Error.</h1>
<h2 class="text-danger">An error occurred while processing your

```

```

request.</h2>

@if (Model.ShowRequestId)
{
<p>
<strong>Request ID:</strong> <code>@Model.RequestId</code>
</p>
}

<h3>header content</h3>
<p>text content</p>

```

In the MVC template's Home Controller, the **Error()** action method sets the **RequestId** to the current **Activity.Current.Id** if available or else it uses **HttpContext.TraceIdentifier**. These values can be useful during debugging.

```

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
public IActionResult Error()
{
    return View(new ErrorViewModel {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier
    });
}

```

## UPDATE:

Wait... what about **Web API** in ASP .NET Core? After posting this article in a popular ASP .NET Core group on Facebook, I got some valuable feedback from the admin:

**Dmitry Pavlov:** "For APIs there is a nice option to handle errors globally with the custom middleware <https://code-maze.com/global-error-handling-aspnetcore> – helps to get rid of try/catch-es in your code. Could be used together with FluentValidation and MediatR – you can configure mapping specific exception types to appropriate status codes (400 bad response, 404 not found, and so on to make it more user friendly and avoid using 500 for everything)."

For more information on the aforementioned items, check out the following resources:

- Global Error Handling in ASP.NET Core Web API: <https://code-maze.com/global-error-handling-aspnetcore/>

- FluentValidation • ASP.NET Integration: <https://fluentvalidation.net/aspnet>
- MediatR Wiki: <https://github.com/jbogard/MediatR/wiki>
- Using MediatR in ASPNET Core Apps: <https://ardalis.com/using-mediatr-in-aspnet-core-apps>

Later on in this series, we'll cover ASP .NET Core Web API in more detail, when we get to "W is for Web API". Stay tuned!

## Error Handling for Razor Pages

Since Razor Pages still use the MVC middleware pipeline, the exception handling is similar to the scenarios described above. For starters, here's what the **Configure()** method looks like in the Startup.cs file for the Razor Pages web app sample.

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Error");
    ...
}
```

In the above code, you can see that the development environment uses the same **DeveloperExceptionPage** middleware. This can be customized using the same techniques outlined in the previous section for MVC pages, so we won't go over this again.

As for the non-dev scenario, the exception handler is slightly different for Razor Pages. Instead of pointing to the **Home** controller's **Error()** action method (as the MVC version does), it points to the to the **/Error** page route. This Error.cshtml Razor Page found in the root level of the **Pages** folder.

```
@page
@model ErrorModel
 @{
    ViewData["Title"] = "Error";
}

<h1 class="text-danger">Error.</h1>
```

```

<h2 class="text-danger">An error occurred while processing your
request.</h2>

@if (Model.ShowRequestId)
{
    <p>
        <strong>Request ID:</strong> <code>@Model.RequestId</code>
    </p>
}

<h3>custom header text</h3>
<p>custom body text</p>

```

The above Error page looks almost identical to the Error view we saw in the previous section, with some notable differences:

- **@page** directive (required for Razor Pages, no equivalent for MVC view)
- uses **ErrorModel** (associated with Error page) instead of ErrorViewModel (served by Home controller's action method)

## Logging Errors

To log errors in ASP .NET Core, you can use the built-in logging features or 3rd-party logging providers. In ASP .NET Core 2.x, the use of **CreateDefaultBuilder()** in Program.cs takes care of default Logging setup and configuration (*behind the scenes*).

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
```

**NOTE:** The Web Host Builder is being replaced by the Generic Host Builder in ASP .NET Core 3.0, but you can expect similar initial behavior. For more information on Generic Host Builder, take a look at the previous blog post in this series: Generic Host Builder in ASP .NET Core.

The host sets up the logging configuration for you, e.g.:

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConfiguration(hostingContext.Configuration.GetSection("Logg
```

```
ing") );
logging.AddConsole();
logging.AddDebug();
logging.AddEventSourceLogger();
})
```

Since ASP .NET Core is open-source, you can find the above snippet (or something similar) on Github. Here is a link to the 2.2 release of `WebHost`:

- `WebHost.cs` at release/2.2:  
<https://github.com/aspnet/MetaPackages/blob/release/2.2/src/Microsoft.AspNetCore/WebHost.cs>

To make use of error logging (in addition to other types of logging) in your MVC web app, you may call the necessary methods in your controller's action methods. Here, you can log various levels of information, warnings and errors at various severity levels.

As seen in the snippet below, you have to do the following in your **MVC Controller** that you want to add Logging to:

1. Add using statement for Logging namespace
2. Add a private readonly variable for an `ILogger` object
3. Inject an `ILogger<model>` object into the constructor
4. Assign the private variable to the injected variable
5. Call various log logger methods as needed.

```
...
using Microsoft.Extensions.Logging;

public class MyController: Controller
{
    ...
    private readonly ILogger _logger;

    public MyController(..., ILogger<MyController> logger)
    {
        ...
        _logger = logger;
    }
}
```

```

public IActionResult MyAction(...)
{
    _logger.LogTrace("log trace");
    _logger.LogDebug("log debug");
    _logger.LogInformation("log info");
    _logger.LogWarning("log warning");
    _logger.LogError("log error");
    _logger.LogCritical("log critical");
}

```

In Razor Pages, the logging code will go into the Page's corresponding Model class. As seen in the snippet below, you have to do the following to the **Model class that corresponds to a Razor Page**:

1. Add using statement for Logging namespace
2. Add a private readonly variable for an ILogger object
3. Inject an ILogger<model> object into the constructor
4. Assign the private variable to the injected variable
5. Call various log logger methods as needed.

```

...
using Microsoft.Extensions.Logging;

public class MyPageModel: PageModel
{
    ...
    private readonly ILogger _logger;

    public MyPageModel(..., ILogger<MyPageModel> logger)
    {
        ...
        _logger = logger;
    }
    ...
    public void MyPageMethod()
    {
        ...
        _logger.LogInformation("log info");
        _logger.LogError("log error");
        ...
    }
}

```

You may have noticed that Steps 1 through 5 are pretty much identical for MVC and Razor Pages. This makes it very easy to quickly add all sorts of logging into your application, including error logging.

## Transient fault handling

Although it's beyond the scope of this article, it's worth mentioning that you can avoid transient faults (e.g. temporary database connection losses) by using some proven patterns, practices and existing libraries. To get some history on transient faults, check out the following article from the classic "patterns & practices". It describes the so-called "Transient Fault Handling Application Block".

- Classic Patterns & Practices: [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn440719\(v=pandp.60\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn440719(v=pandp.60))

More recently, check out the docs on Transient Fault Handling:

- Docs: <https://docs.microsoft.com/en-us/aspnet/aspnet/overview/developing-apps-with-windows-azure/building-real-world-cloud-apps-with-windows-azure/transient-fault-handling>

And now in .NET Core, you can add resilience and transient fault handling to your .NET Core HttpClient with Polly!

- Adding Resilience and Transient Fault handling to your .NET Core HttpClient with Polly: <https://www.hanselman.com/blog/AddingResilienceAndTransientFaultHandlingToYourNETCoreHttpClientWithPolly.aspx>
- Integrating with Polly for transient fault handling: <https://www.stevejgordon.co.uk/httpclientfactory-using-polly-for-transient-fault-handling>

You can get more information on the Polly project on the official Github page:

- Polly on Github: <https://github.com/App-vNext/Polly>

# References

- Read text from a file: <https://docs.microsoft.com/en-us/dotnet/standard/io/how-to-read-text-from-a-file>
- try-catch-finally: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch-finally>
- Handle errors in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/error-handling>
- Use multiple environments in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments>
- UseDeveloperExceptionPage: <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.developerexceptionpageextensions.usedeveloperexceptionpage>
- DeveloperExceptionPageOptions: <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.developerexceptionpageoptions>
- 2.x Sample: <https://github.com/aspnet/Docs/tree/master/aspnetcore/fundamentals/error-handling/samples/2.x/ErrorHandlingSample>
- Razor  
Sample: [https://github.com/shahedc/\\_workshops/tree/master/AspNetCoreRazor/AspNetCoreRazor](https://github.com/shahedc/_workshops/tree/master/AspNetCoreRazor/AspNetCoreRazor)
- MVC  
Sample: [https://github.com/shahedc/\\_workshops/tree/master/AspNetCoreMvc/MvcMovie](https://github.com/shahedc/_workshops/tree/master/AspNetCoreMvc/MvcMovie)
- Logging: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging>

# IIS Hosting for ASP .NET Core Web Apps

By Shahed C on March 4, 2019

1 Reply

This is the **ninth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

- I is for IIS Hosting
- Developing for IIS
- IIS Configuration in Visual Studio
- ASP .NET Core Module
- BONUS: Publishing to a VM with IIS
- References

## I is for IIS Hosting

If you've been reading my weekly blog posts in this series (or you've worked with ASP .NET Core), you probably know that ASP .NET Core web apps can run on multiple platforms. Since Microsoft's IIS (Internet Information Services) web server only runs on Windows, you may be wondering why we would need to know about IIS-specific hosting at all.

Well, we don't *need* IIS to run ASP .NET Core, but there are some useful IIS features you can take advantage of. In fact, ASP .NET Core v2.2 introduced in-process hosting in IIS with the ASP .NET Core module. You can run your app either in-process or out of process.

From the 2.2 release notes of what's new:

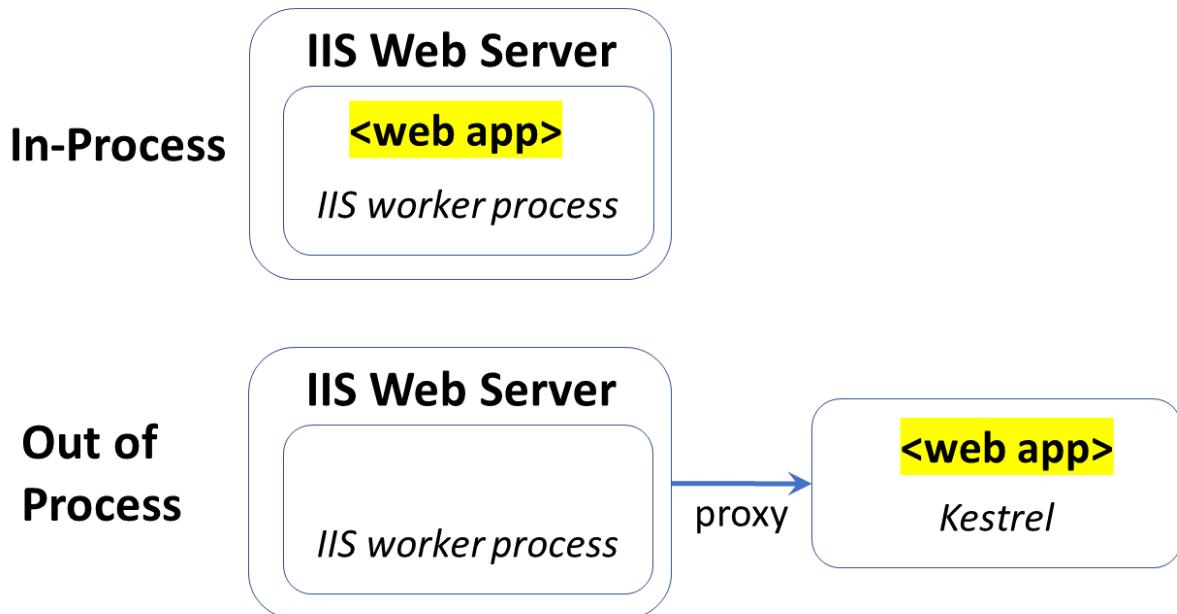
*"In earlier versions of ASP.NET Core, IIS serves as a reverse proxy. In 2.2, the ASP.NET Core Module can boot the CoreCLR and host an app inside the IIS worker process (w3wp.exe). In-process hosting provides performance and diagnostic gains when running with IIS."*

You can browse an IIS-configured sample app on GitHub at the following location:



IIS-configured sample: <https://github.com/shahedc/IISHostedWebApp>

**NOTE:** The actual web.config file has been intentionally left out from the above repo, and replaced with a reference file, web.config.txt. When you follow the configuration steps outlined below, the actual web.config file will be generated with the proper settings.



## Developing for IIS

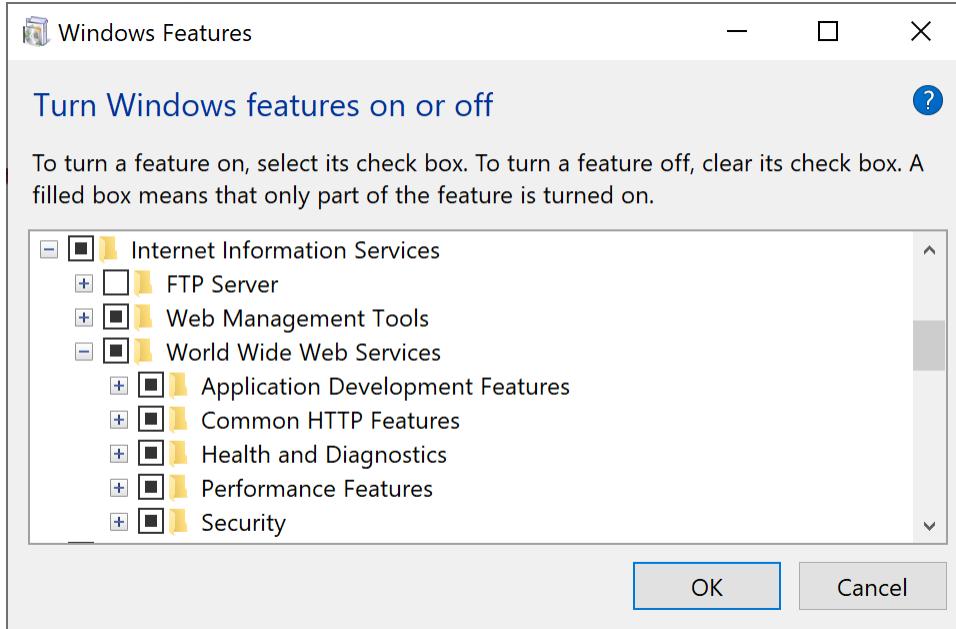
In order to develop for IIS on your local development machine while working with Visual Studio, you should complete the following steps:

1. **Install/Enable IIS locally:** add IIS via Windows Features setup
2. **Configure IIS:** add website with desired port(s)
3. **Enable IIS support in VS:** include dev-time IIS support during setup
4. **Configure your web app:** enable HTTPS, add launch profile for IIS

**NOTE:** If you need help with any of the above steps, you may follow the detailed guide in the official docs:

- Development-time IIS support in Visual Studio for ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/development-time-iis-support?view=aspnetcore-2.2>

**After Step 1 (IIS installation)**, the list of Windows Features should show that IIS and many of its components have been installed.



**During and after Step 2 (IIS configuration)**, my newly added website looks like the screenshots shown below:

Add Website

Site name: **Dev Website** Application pool: **Dev Website** [Select...](#)

Content Directory

Physical path: **C:\webdev** [...](#)

Pass-through authentication

[Connect as...](#) [Test Settings...](#)

Binding

Type: **https** IP address: **All Unassigned** Port: **443**

Host name: **localhost**

Require Server Name Indication

Disable HTTP/2

Disable OCSP Stapling

SSL certificate: **IIS Express Development Certificate** [Select...](#) [View...](#)

Start Website immediately

**OK** **Cancel**

Add New Website

Internet Information Services (IIS) Manager

Sites > Dev Web Site > IISHostedWebApp

File View Help

Connections

Application Pools

Sites

Default Web Site

Dev Web Site

IISHostedWebApp

bin  
obj  
Pages  
Properties  
wwwroot

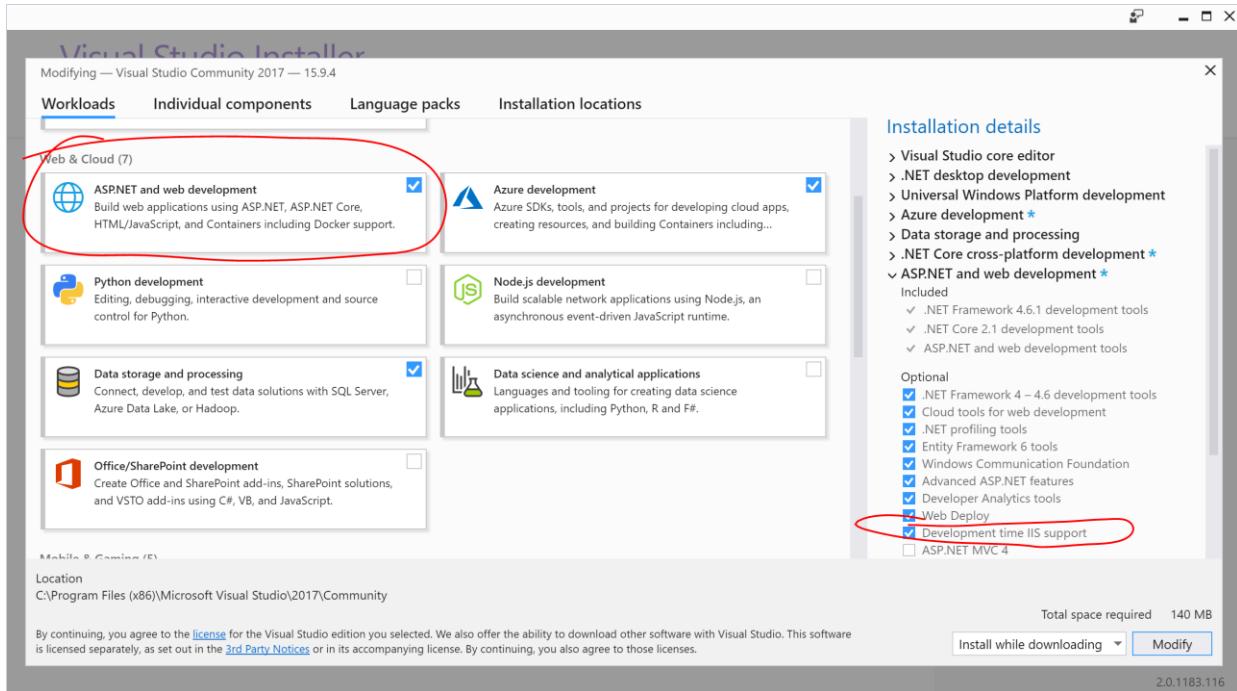
/IISHostedWebApp Content

Filter: Go Show All Group by: No Group

Name	Type
bin	File Folder
obj	File Folder
Pages	File Folder
Properties	File Folder
wwwroot	File Folder
appsettings.Development.json	json_auto_file
appsettings.json	json_auto_file
IISHostedWebApp.csproj	Visual C# Project file
IISHostedWebApp.csproj.user	Per-User Project Options File
Program.cs	CS File
Startup.cs	CS File
web.config	CONFIG File

Explore Added Website

**For Step 3 (IIS support in VS) mentioned above, make sure you select the “Development time IIS support” option under the “ASP .NET and web development” workload. These options are shown in the screenshot below:**



**After Step 4** (web app config), I can run the web application locally on IIS. Check out the next section to learn more about your web application configuration. In addition to the .csproj settings, you'll also need a web.config file in addition to your appsettings.json file.

*Optional*, you could set up a VM on Azure with IIS enabled and deploy to that web server. If you need some help with the setup, you can select a pre-configured Windows VM with IIS pre-installed from the Azure Portal.

If you need help with creating an Azure VM with IIS, check out the following resources:

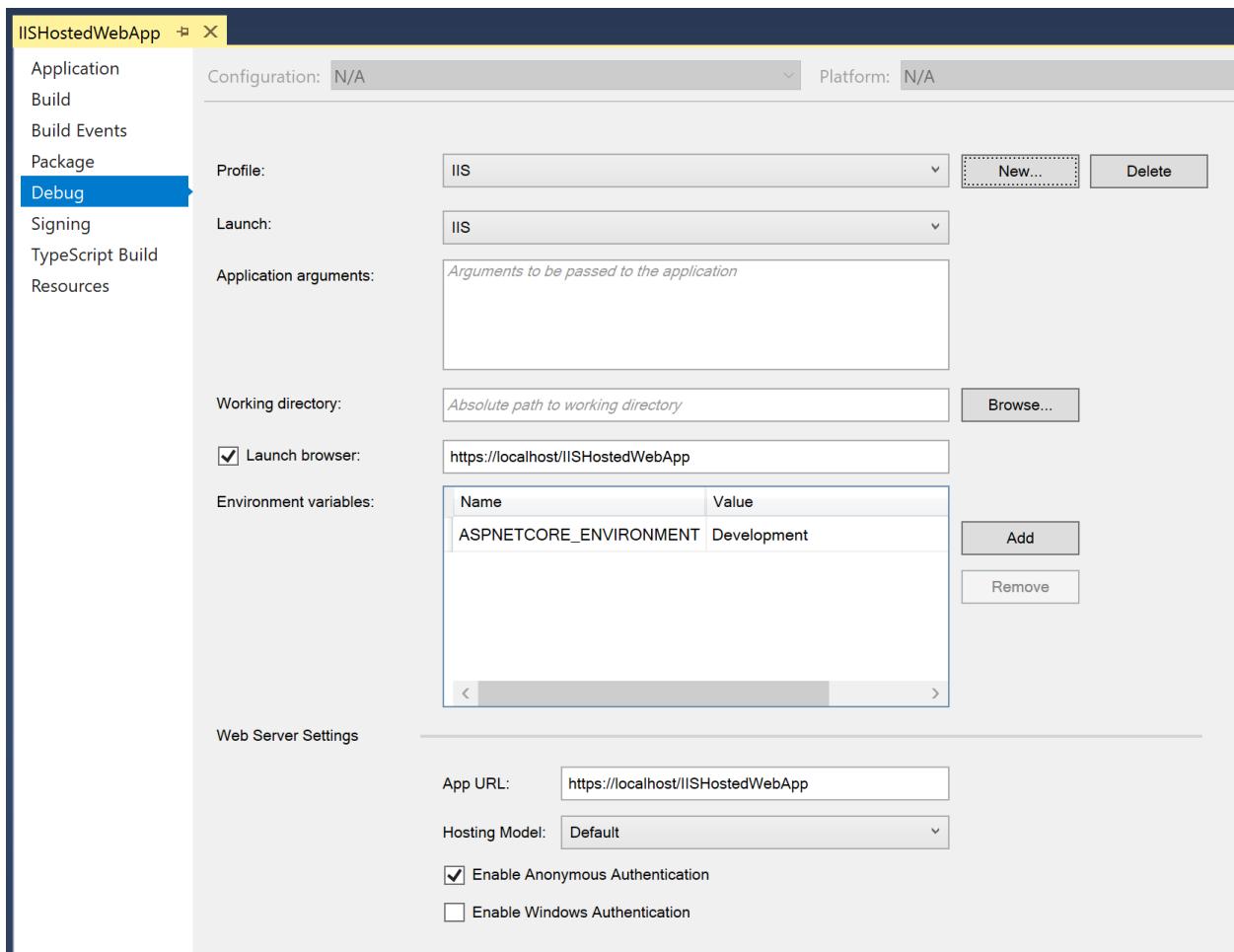
- Create Windows VM in Azure portal: <https://docs.microsoft.com/en-us/azure/virtual-machines/windows/quick-create-portal>
- Create VMs running an IIS, etc: <https://docs.microsoft.com/en-us/azure/virtual-machines/windows/tutorial-iis-sql>
- (Pre-configured) IIS on Windows Server 2016: <https://azuremarketplace.microsoft.com/en-us/marketplace/apps/apps-4-rent.iis-on-windows-server-2016>

# IIS Configuration in Visual Studio

After completing Step 4 using the instructions outlined above, let's observe the following:

- **Profile:** Debug tab in project properties
- **Project:** .csproj settings
- **Settings:** launchSettings.json
- **Config:** web.config file

**Profile:** With your web project open in Visual Studio, right-click the project in Solution Explorer, and then click **Properties**. Click the Debug tab to see the newly-added IIS-specific profile. Note that the Hosting Model offers 3 options: Default, In-Process and Out-of-process.



**Project:** Again in the Solution Explorer, right-click the project, then click Edit `<projname>.csproj` to view the .csproj file as a text file. Here, the `<AspNetCoreHostingModel>` setting is shown set to **InProcess**, and can be changed to **OutOfProcess** if desired.

The screenshot shows the Visual Studio interface with the 'IISHostedWebApp.csproj' file open in the code editor. The code is a .NET Core project file (csproj) with the following content:

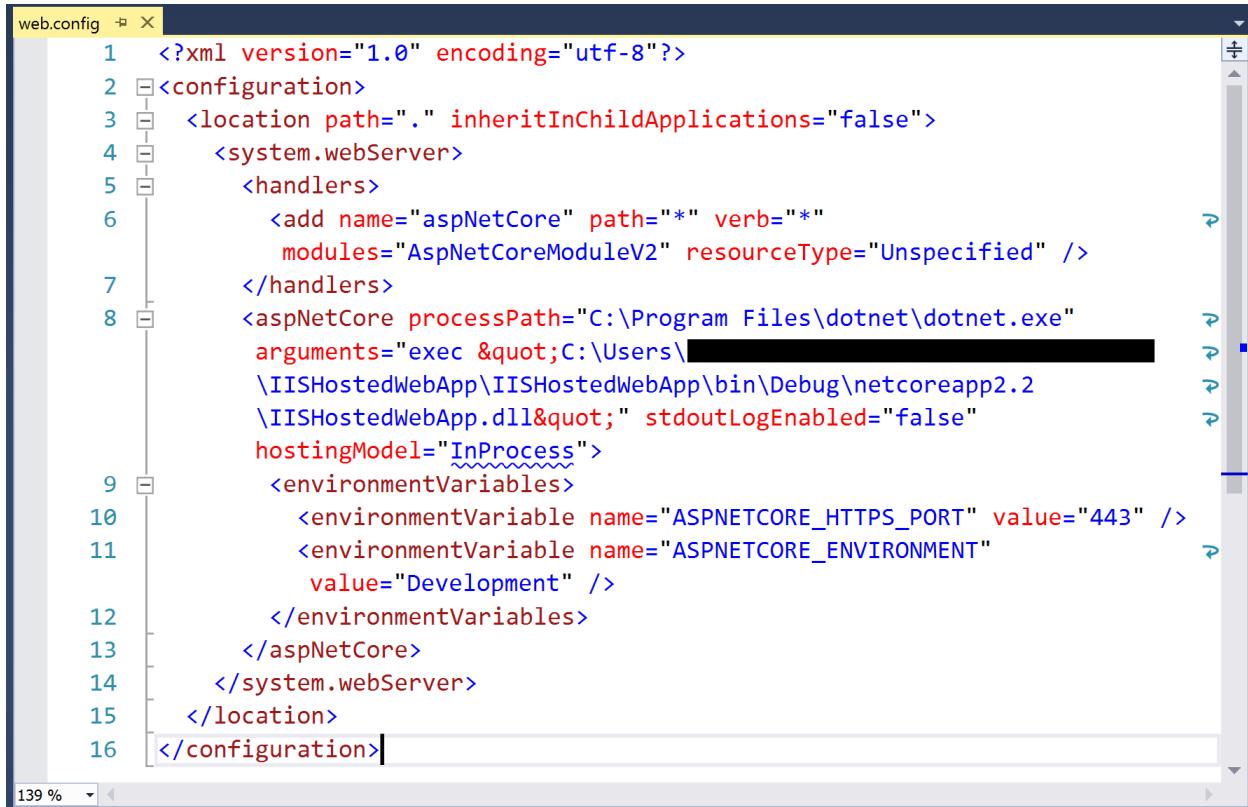
```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
  </PropertyGroup>
  <ItemGroup>
    <None Remove="web.config.txt" />
  </ItemGroup>
  <ItemGroup>
    <Content Include="web.config.txt" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0" PrivateAssets="All" />
  </ItemGroup>
</Project>
```

**Settings:** Under the Properties node in the Solution Explorer, open the *launchSettings.json* file. Here, you can see all the settings for your newly-created IIS-specific Profile. You can manually add/edit settings directly in this file without having to use the Visual Studio UI.



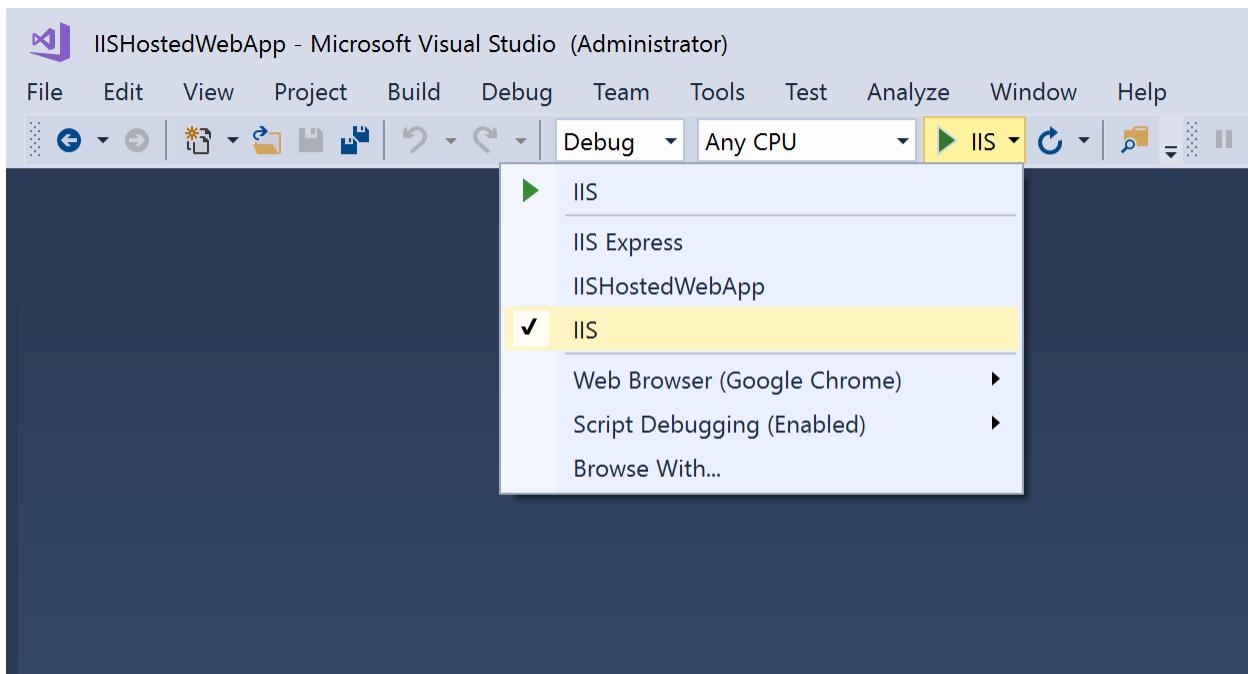
```
1  {
2   "iisSettings": {
3     "windowsAuthentication": false,
4     "anonymousAuthentication": true,
5     "iis": {
6       "applicationUrl": "https://localhost/IISHostedWebApp",
7       "sslPort": 0
8     },
9     "iisExpress": [...]
13   },
14   "profiles": {
15     "IIS Express": [...],
16     "IISHostedWebApp": [...],
17     "IIS": {
18       "commandName": "IIS",
19       "launchBrowser": true,
20       "launchUrl": "https://localhost/IISHostedWebApp",
21       "environmentVariables": {
22         "ASPNETCORE_ENVIRONMENT": "Development"
23       }
24     }
25   }
26 }
```

**Config:** You may recognize the XML-based web.config file from previous versions of ASP .NET, but the project's settings and dependencies are now spread across multiple files, such as the .csproj file and appsettings.json config file. For ASP .NET Core projects, the web.config file is only used specifically for an IIS-hosted environment. In this file, you may configure the ASP .NET Core Module that will be used for your IIS-hosted web application.

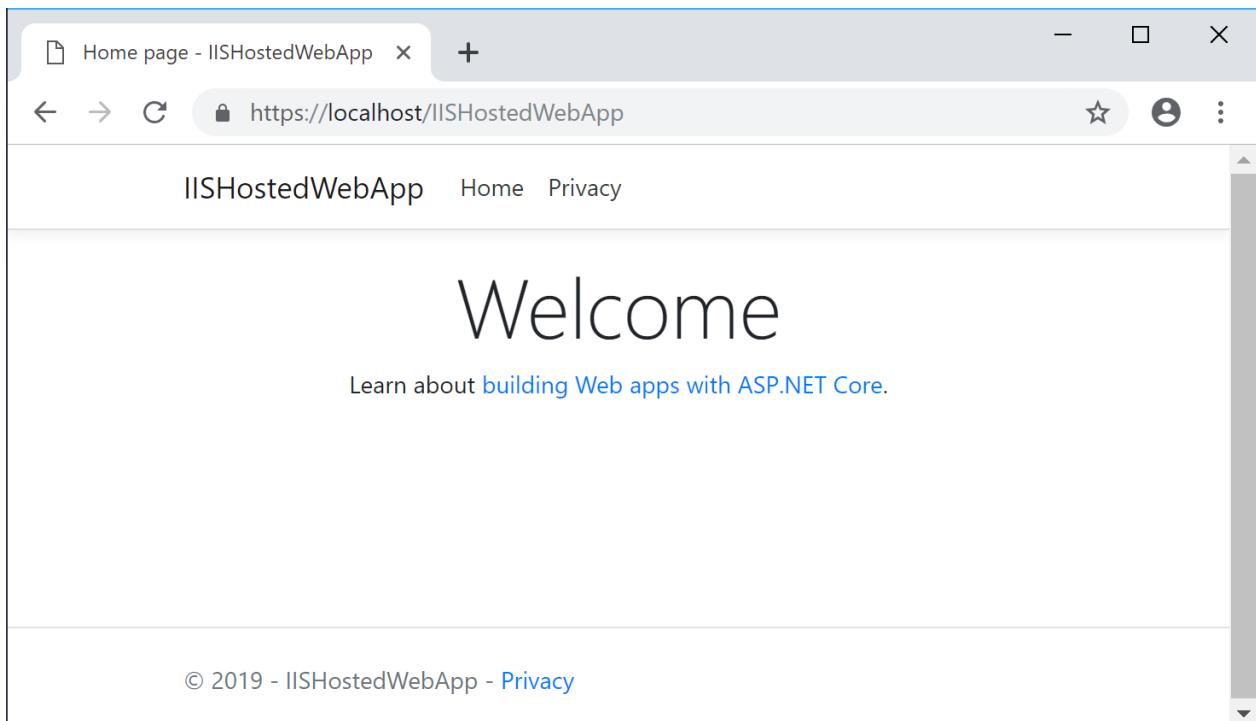


```
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3    <location path=".\" inheritInChildApplications="false">
4      <system.webServer>
5        <handlers>
6          <add name="aspNetCore" path="*" verb="*"
7            modules="AspNetCoreModuleV2" resourceType="Unspecified" />
8        </handlers>
9        <aspNetCore processPath="C:\Program Files\dotnet\dotnet.exe"
10          arguments="exec "C:\Users\██████████\IISHostedWebApp\IISHostedWebApp\bin\Debug\netcoreapp2.2
11          \IISHostedWebApp.dll"" stdoutLogEnabled="false"
12          hostingModel="InProcess">
13          <environmentVariables>
14            <environmentVariable name="ASPNETCORE_HTTPS_PORT" value="443" />
15            <environmentVariable name="ASPNETCORE_ENVIRONMENT"
16              value="Development" />
17          </environmentVariables>
18        </aspNetCore>
19      </system.webServer>
20    </location>
21  </configuration>
```

Once your web app is configured, you can run it from Visual Studio using the profile you created previously. In order to run it using a specific profile, click the tiny dropdown next to the green Run/Debug button to select the desired profile. Then click the button itself. This should launch the application in your web browser, pointing to localhost running on IIS, e.g. <https://localhost/<appname>>



IIS Profile



Running in a browser

# ASP .NET Core Module

Now that your IIS server environment has been set up (either locally or in the cloud), let's learn about the ASP .NET Core Module. This is a native IIS module that plugs into the actual IIS pipeline (not to be confused with your web app's request pipeline). It allows you to host your web app in one of two ways:

- **In-process:** the web app is hosted *inside* the IIS worker process, i.e. w3wp.exe (previously known as the World Wide Web Publishing Service)
- **Out of process:** IIS *forwards* web server requests to the web app, which uses Kestrel (the cross-platform web server included with ASP .NET Core)

As you've seen in an earlier section, this setting is configured in the ***web.config*** file:

```
<PropertyGroup>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

The lists below shows some notable differences between hosting in-process vs out of process.

## In-Process

- IISHttpServer within IIS
- Module waits for app to process request
- 1 app pool per app
- `app_offline.htm` may not cause immediate shutdown

## Out of Process

- Kestrel outside IIS via proxy
- Module waits for response from process listening on port
- App pool sharing allowed
- `app_offline.htm` causes immediate shutdown

How you configure your app is up to you. For more details on the module and its configuration settings, browse through the information available at:

- ASP.NET Core Module: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/aspnet-core-module>

## BONUS: Publishing to a VM with IIS

You can publish your ASP .NET Core web app to a Virtual Machine (either on your network or in Azure) or just any Windows Server you have access to. There are several different options:

- Deploy an ASP.NET app to an Azure virtual machine: <https://tutorials.visualstudio.com/aspnet-vm/intro>
- Publish a Web App to an Azure VM from Visual Studio: <https://docs.microsoft.com/en-us/azure/virtual-machines/windows/publish-web-app-from-visual-studio>
- Deploy your ASP.NET app to Azure virtual machines by using Azure DevOps Projects: <https://docs.microsoft.com/en-us/azure/devops-project/azure-devops-project-vms>

Here are some prerequisites to be aware of:

- IIS must be pre-installed on the server with relevant ports enabled
- WebDeploy must be installed (which you would normally have on your local dev machine)
- The VM must have a DNS name configured (Azure VMs can have fully qualified domain names, e.g. <machinename>.eastus.cloudapp.azure.com)

## References

- IIS in-process hosting in ASP.NET Core 2.2: <https://docs.microsoft.com/en-us/aspnet/core/release-notes/aspnetcore-2.2?view=aspnetcore-2.2#iis-in-process-hosting>
- Development-time IIS support in Visual Studio for ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/development-time-iis-support>

- Host ASP.NET Core on Windows with IIS: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/>
- ASP.NET Core Module: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/aspnet-core-module>
- IIS modules with ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/modules>
- Hosting ASP.NET Core Applications on IIS: <https://codewala.net/2019/02/05/hosting-asp-net-core-applications-on-iis-in-process-hosting/>

# JavaScript, CSS, HTML & Other Static Files in ASP .NET Core

By Shahed C on March 11, 2019

3 Replies

This is the **tenth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



**A – Z of ASP .NET Core!**

## In this Article:

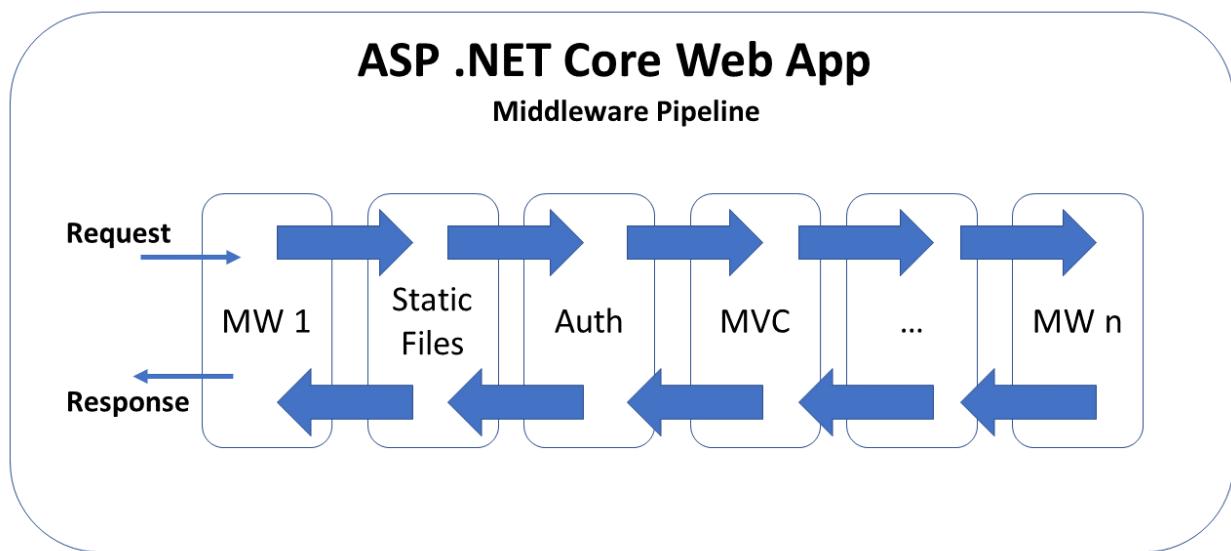
- J is for JavaScript, CSS, HTML & Other Static Files
- Configuring Static Files via Middleware
- Customizing Locations for Static Files
- Preserving CDN Integrity
- LibMan (aka Library Manager)
- What About NPM or WebPack?
- References

# J is for JavaScript, CSS, HTML & Other Static Files

**NOTE:** this article will teach you how to include and customize the use of static files in ASP .NET Core web applications. It is not a tutorial on front-end web development.

If your ASP .NET Core web app has a front end – whether it's a collection of MVC Views or a Single-Page Application (SPA) – you will need to include static files in your application. This includes (but is not limited to): JavaScript, CSS, HTML and various image files.

When you create a new web app using one of the built-in templates (MVC or Razor Pages), you should see a “wwwroot” folder in the Solution Explorer. This points to a physical folder in your file system that contains the same files seen from Visual Studio. However, this location can be configured, you can have multiple locations with static files, and you can enable/disable static files in your application if desired. In fact, you have to “opt in” to static files in your middleware pipeline.



You can browse the (template-generated) sample app (with static files) on GitHub at the following location:



sample app with static files: <https://github.com/shahedc/AspNetCoreStaticSample>

## Configuring Static Files via Middleware

Let's start by observing the `Startup.cs` configuration file. We've seen this file several times throughout this blog series. In the `Configure()` method, you'll find the familiar method call to enable the use of static files.

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

This call to `app.UseStaticFiles()` ensures that static files can be served from the designated location, e.g. `wwwroot`. In fact, this line of code looks identical whether you look at the `Startup.cs` file in an MVC or Razor Pages project.

It's useful to note the placement of this line of code. It appears *before* `app.UseMvc()`, which is very important. This ensures that static file requests can be processed and sent back to the web browser without having to touch the MVC middleware. This becomes even more important when authentication is used.

Take a look at either the MVC or Razor Pages projects that have authentication added to them. In the code below, you can see the familiar call to `app.UseStaticFiles()` once again. However, there is also a call to `app.UseAuthentication()`. It's important for the authentication call to appear *after* the call to use static files. This ensure that the authentication process isn't triggered when it isn't needed.

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

```
    app.UseAuthentication();
    ...
    app.UseMvc(...);
}
```

By using the middleware pipeline in this way, you can “short-circuit” the pipeline when a request has been fulfilled by a specific middleware layer. If a static file has been successfully served using the Static Files middleware, it prevents the next layers of middleware (i.e. authentication, MVC) from processing the request.

**NOTE:** if you need to secure any static files, e.g. private images, you can consider a cloud solution such as Azure Blob Storage to store the files. The files can then be served from within the application, instead of actual static files. You could also serve secure files (from outside the wwwroot location) as a `FileResult()` object returned from an action method that has an `[Authorize]` attribute.

## Customizing Locations for Static Files

It may be convenient to have the default web templates create a location for your static files and also enable the use of those static files. As you’ve already seen, enabling static files isn’t magic. Removing the call to `app.useStaticFiles()` will disable static files from being served. In fact, the location for static files isn’t magic either.

In a previous post, we had discussed how the `Program.cs` file includes a call to `CreateDefaultBuilder()` to set up your application:

```
public class Program
{
    ...
    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
=>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
}
```

Behind the scenes, this method call sets the “content root” to the current directory, which contains the “wwwroot” folder, your project’s “web root”. These can both be customized.

- To change the content root, you can configure it while building the host, e.g.

```
WebHost.CreateDefaultBuilder(args).UseContentRoot("c:\\<content-root>")
```

- To change the web root within your content root, you can also configure it while building the host, e.g.

```
WebHost.CreateDefaultBuilder(args).UseWebRoot("public")
```

You may also use the call to **app.UseStaticFiles()** to customize an alternate location to serve static files. This allows you to serve additional static files from a location outside of the designated web root.

```
...
using Microsoft.Extensions.FileProviders;
using System.IO;
...
public void Configure(IApplicationBuilder app)
{
    ...
    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(env.ContentRootPath, "AltStaticRoot")) ,
        RequestPath = "/AltStaticFiles"
    ) ;
}
```

**Wait a minute...** why does it look like there are *two* alternate locations for static files? There is a simple explanation:

- In the call to **Path.Combine()**, the "**AltStaticRoot**" is an actual folder in your current directory. This Path class and its Combine() method are available in the System.IO namespace.
- The "**AltStaticFiles**" value for **RequestPath** is used as a root-level "virtual folder" from which images can be served. The **PhysicalFileProvider** class is available in the Microsoft.Extensions.FileProviders namespace.

The following markup may be used in a .cshtml file to refer to an image, e.g. MyImage01.png:

```

```

The screenshot below shows an example of an image loaded from an alternate location.

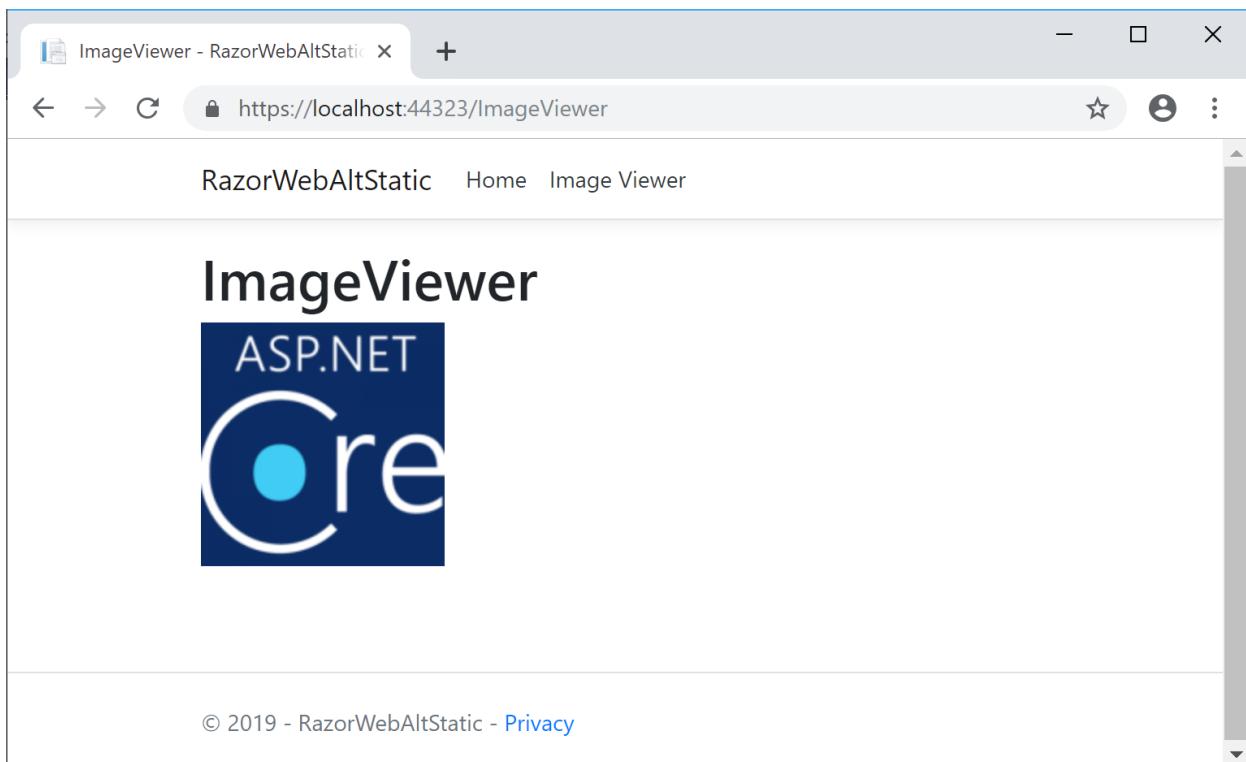
The screenshot shows the Visual Studio IDE interface. On the left, the code editor displays the file `ImageViewer.cshtml` with the following content:

```
1 @page
2 @{
3     ViewData["Title"] = "ImageViewer";
4 }
5
6 <h1>ImageViewer</h1>
7
8 <div>
9     
10 </div>
```

The Solution Explorer window on the right shows the project structure for `RazorWebAltStatic`:

- Connected Services
- Dependencies
- Properties
- wwwroot
  - AltStaticRoot
    - MyImages
      - MyImage01.png
  - Pages
    - Shared
      - \_ViewImports.cshtml
      - \_ViewStart.cshtml
    - ImageViewer.cshtml
    - Index.cshtml
    - Privacy.cshtml

The screenshot below shows a web browser displaying such an image.



You can find the above code in the `RazorWebAltStatic` sample web app on GitHub:

- <https://github.com/shahedc/AspNetCoreStaticSample/tree/master/RazorWebAltStatic>

# Preserving CDN Integrity

When you use a CDN (Content Delivery Network) to serve common CSS and JS files, you need to ensure that the integrity of the source code is reliable. You can rest assured that ASP .NET Core has already solved this problem for you in its built-in templates. Let's take a look at the shared Layout page, e.g. `_Layout.cshtml` in a Razor Pages web application.

```
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css"
/>
</environment>

<environment exclude="Development">
  <link
    rel="stylesheet"
    href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/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="absolute"
    crossorigin="anonymous"
    integrity="sha256-eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE="
  </environment>
```

Right away, you'll notice that there are two conditional `<environment>` blocks in the above markup. The first block is used only during development, in which the bootstrap CSS file is obtained from your local copy. When *not* in development (e.g. staging, production, etc), the bootstrap CSS file is obtained from a CDN, e.g. CloudFlare.

You could use an automated hash-generation tool to generate the SRI (Subresource Integrity) hash values, but you would have to manually copy the value into your code. You can try out the relatively-new LibMan (aka Library Manager) for easily adding and updating your client-side libraries.

# LibMan (aka Library Manager)

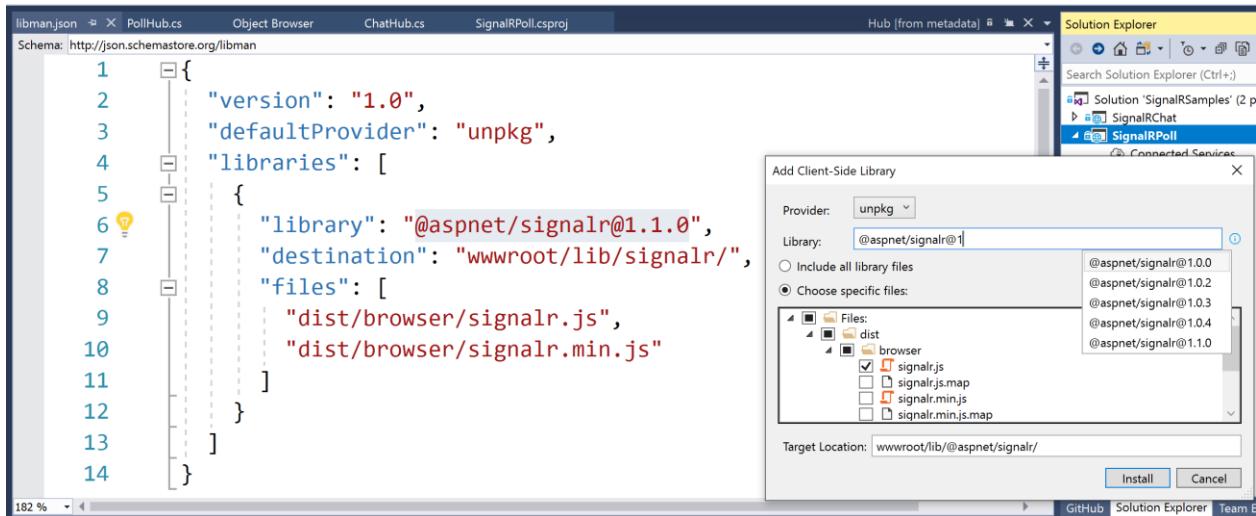
The easiest way to use LibMan is to use the built-in features available in Visual Studio. Using LibMan using the IDE is as easy as launching it from Solution Explorer. Specify the provider from the library you want, and any specific files you want from that library.

If you've already read my SignalR article from my 2018 blog series, you may recall the steps to add a client library via LibMan (aka Library Manager):

- Right-click project in Solution Explorer
- Select Add | Client-Side Library

In the popup that appears, select/enter the following:

- **Provider:** choose from cdnjs, filesystem, unpkg
- **Library** search term, e.g. @aspnet/signalr@1... pick latest stable if desired
- **Files:** At a minimum, choose specific files, e.g. signalr.js and/or its minified equivalent



For more on LibMan (using VS or CLI), check out the official docs:

- Use LibMan with ASP.NET Core in Visual Studio: <https://docs.microsoft.com/en-us/aspnet/core/client-side/libman/libman-vs>

- Use the LibMan command-line interface (CLI): <https://docs.microsoft.com/en-us/aspnet/core/client-side/libman/libman-cli>
- Library Manager: Client-side content manager for web apps: <https://devblogs.microsoft.com/aspnet/library-manager-client-side-content-manager-for-web-apps/>

In any case, using LibMan will auto-populate a “***libman.json***” manifest file, which you can also inspect and edit manually. The aforementioned SignalR article also includes a real-time polling web app sample. You can view its libman.json file to observe its syntax for using a SignalR client library.

```
{
  "version": "1.0",
  "defaultProvider": "unpkg",
  "libraries": [
    {
      "library": "@aspnet/signalr@1.1.0",
      "destination": "wwwroot/lib/signalr/",
      "files": [
        "dist/browser/signalr.js",
        "dist/browser/signalr.min.js"
      ]
    }
  ]
}
```

Every time you save the libman.json file, it will trigger LibMan’s restore process. This pulls down the necessary libraries from their specified source, and adds them to your local file system. If you want to trigger this restore process manually, you can always choose the “Restore Client-Side Libraries” option by right-clicking the libman.json file in Solution Explorer.

## What About NPM or WebPack?

If you’ve gotten this far, you may be wondering: “*hey, what about NPM or WebPack?*”

It’s good to be aware that LibMan is not a replacement for your existing package management systems. In fact, the Single-Page Application templates in Visual Studio (for Angular and React) currently use npm and WebPack. LibMan simply provides a lightweight mechanism to include client-side libraries from external locations.

For more information on WebPack in ASP .NET Core, I would recommend these 3rd-party articles:

- ReactJs Webpack and ASP.NET Core: <https://sensibledev.com/reactjs-webpack-and-asp-net-core/>
- Updating Your JavaScript Libraries in ASP.NET Core 2.2 (*includes LibMan, NPM, WebPack*): <https://medium.com/@scottkuhl/updating-your-javascript-libraries-in-asp-net-core-2-2-3c2d985a491e>

## References

- Static files in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files>
- ASP.NET Core Middleware: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/>
- Library Manager (LibMan): <https://devblogs.microsoft.com/aspnet/library-manager-client-side-content-manager-for-web-apps/>
- Use LibMan with ASP.NET Core in Visual Studio: <https://docs.microsoft.com/en-us/aspnet/core/client-side/libman/libman-vs>
- Get started with ASP.NET Core SignalR: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr>
- Environment Tag Helper in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/environment-tag-helper>
- Securing the CDN links in the ASP.NET Core 2.1 templates: <https://damienbod.com/2018/03/14/securing-the-cdn-links-in-the-asp-net-core-2-1-templates/>
- Pre-compressed static files with ASP.NET Core: <https://gunnarpeipman.com/aspnet/pre-compressed-files/>

Additional reference for pre-compressing static files, found on Twitter via:

@DotNetAppDev: <https://twitter.com/DotNetAppDev/status/1106299093922533376>

Pre-compressed static files with <https://t.co/okVPKPWa3e> Core <https://t.co/Keeczwbggs>

— .NET App Dev News (@DotNetAppDev) March 14, 2019

# Key Vault for ASP .NET Core Web Apps

By Shahed C on March 18, 2019

7 Replies

This is the **eleventh** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

- K is for Key Vault for ASP .NET Core Web Apps
- Setting up Key Vault in Azure
- Retrieving Key Vault Secrets
- Managed Service Identity
- References

## K is for Key Vault for ASP .NET Core Web Apps

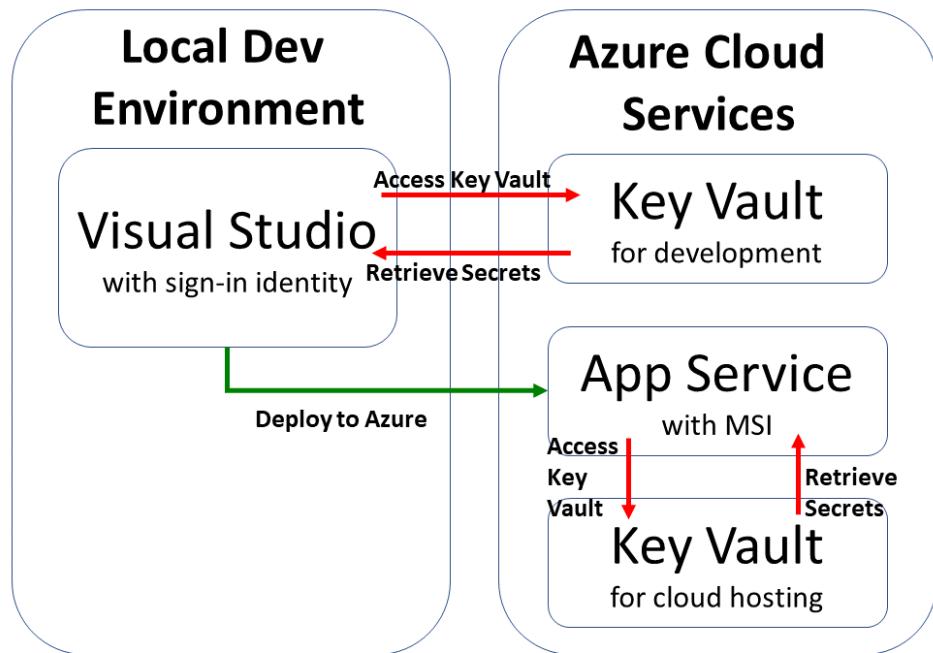
In my 2018 blog series, we covered the use of User Secrets for your ASP .NET Core web application projects. This is useful for storing secret values locally during development. However, we need a cloud-based scalable solution when deploying web apps to Azure. This article will cover Azure Key Vault as a way to store and retrieve sensitive information in Azure and access them in your web application.

You may download the following sample project to follow along with Visual Studio. You may need to apply migrations if you wish to use the optional authentication features with a data store in the sample app.



Key Vault Sample: <https://github.com/shahedc/AspNetCoreKeyVaultSample>

You will also need an Azure subscription to create and use your own Key Vault and App Service.



## Setting up Key Vault in Azure

Before you can use Key Vault in the cloud, you will have to set it up in Azure for your use. This can be done in various ways:

- **Azure Portal**: log in to the Azure Portal in a web browser.
- **Azure CLI**: use Azure CLI commands on your development machine.
- **Visual Studio**: use the VS IDE on your development machine.

To use the **Azure Portal**: create a new resource, search for Key Vault, click Create and then follow the onscreen instructions. Enter/select values for the following for the key vault:

- *Name*: alphanumeric, dashes allowed, cannot start with number
- *Subscription*: select the desired Azure subscription
- *Resource Group*: select a resource group or create a new one
- *Location*: select the desired location
- *Pricing Tier*: select the appropriate pricing tier (Standard, Premium)
- *Access policies*: select/create one or more access policies with permissions
- *Virtual Network Access*: select all (or specific) networks to allow access

**Key Vault**

**Enhance data protection and compliance**

Secure key management is essential to protecting data in the cloud. With Azure Key Vault, you can safeguard encryption keys and application secrets like passwords using keys stored in hardware security modules (HSMs). For added assurance, you can import or generate your encryption keys in HSMs. If you choose to do this, Microsoft will process your keys in FIPS 140-2 Level 2 validated HSMs (hardware and firmware). Key Vault is designed so that Microsoft does not see or extract your keys. Monitor and audit key use with Azure logging-pipe logs into Azure HDInsight or your SIEM for additional analysis and threat detection.

**All of the control, none of the work**

With Key Vault, there's no need to provision, configure, patch, and maintain HSMs and key management software. You can provision new vaults and keys (or import keys from your own HSMs) in minutes and centrally manage keys, secrets, and policies. You maintain control over your keys—simply grant permission for your own and third-party applications to use them as needed. Applications never have direct access to keys. Developers easily manage keys used for Dev/Test and migrate seamlessly to production keys managed by security operations.

**Boost performance and achieve global scale**

Improve performance and reduce the latency of cloud applications by storing cryptographic keys in the cloud instead of on-premises. Key Vault rapidly scales to meet the cryptographic needs of your cloud applications and match peak demand without the cost associated with deploying dedicated HSMs. You can achieve global redundancy by provisioning vaults in Azure global datacenters—keep a copy in your own HSMs for added durability.

[Save for later](#)

Select a software plan

**Key Vault**  
Safeguard cryptographic keys and other secrets used by cloud apps and services.

[Create](#) [Automation options](#)

If you need help with the Azure Portal, check out the official docs at:

- Set and retrieve a secret from Key Vault using Azure portal: <https://docs.microsoft.com/en-us/azure/key-vault/quick-create-portal>

To use the **Azure CLI**: authenticate yourself, run the appropriate commands to create a key vault, add keys/secrets/certificates and then authorize an application to use your keys/secrets.

To create a new key vault, run “**az keyvault create**” followed by a name, resource group and location, e.g.

```
az keyvault create --name "MyKeyVault" --resource-group "MyRG" --location "East US"
```

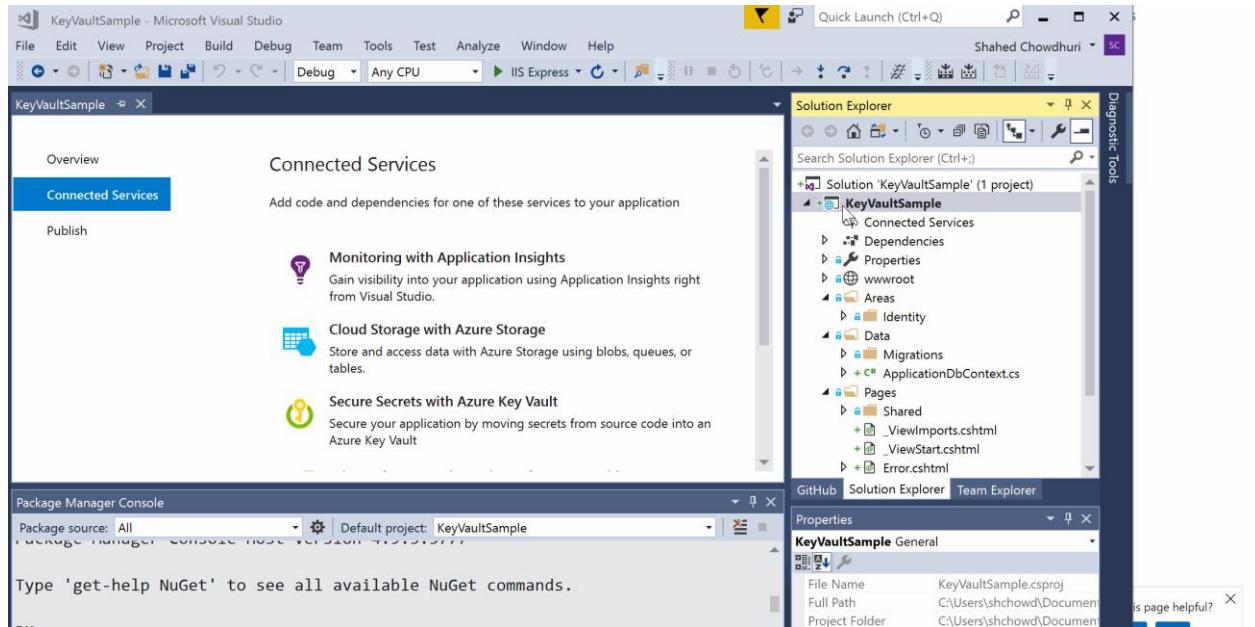
To add a new secret, run “**az keyvault secret set**”, followed by the vault name, a secret name and the secret’s value, e.g.

```
az keyvault secret set --vault-name "MyKeyVault" --name "MySecretName" --value "MySecretValue"
```

If you need help with the Azure CLI, check out the official docs at:

- Manage Azure Key Vault using CLI: <https://docs.microsoft.com/en-us/azure/key-vault/key-vault-manage-with-cli2>

To use **Visual Studio**, right-click your project in Solution Explorer, click Add | Connected Service, select “Secure Secrets with Azure Key Vault” and follow the onscreen instructions.



If you need help with Visual Studio, check out the official docs at:

- Add Key Vault support to your ASP.NET project using Visual Studio: <https://docs.microsoft.com/en-us/azure/key-vault/vs-key-vault-add-connected-service>

Once created, the Key Vault and its secret names (and values) can be browsed in the Azure Portal, as seen in the screenshots below:

The screenshot shows the 'Overview' tab of a Key Vault in the Azure portal. The left sidebar includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Settings', 'Keys', and 'Secrets'. The main pane displays resource details: Resource group (changed), Location, Subscription (changed), Subscription ID, DNS Name (https://[REDACTED].vault.azure.net/), Sku (Pricing tier) Standard, Directory ID, and Directory Name.

### Key Vault Details

The screenshot shows the 'Secrets' tab of the Key Vault in the Azure portal. The left sidebar includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Settings', 'Keys', and 'Secrets'. The main pane shows a success message: 'The secret 'MyKeyVaultSecret' has been successfully created.' A table lists the secret: NAME MyKeyVaultSecret, TYPE secret string value, and STATUS ✓ Enabled.

NAME	TYPE	STATUS	EXPIRATION DATE
MyKeyVaultSecret	secret string value	✓ Enabled	

### Secret in Key Vault

**NOTE:** If you create a secret named “Category1–MySecret”, this syntax specifies a category “Category1” that contains a secret “MySecret.”

# Retrieving Key Vault Secrets

Before you deploy your application to Azure, you can still access the Key Vault using Visual Studio during development. This is accomplished by using your Visual Studio sign-in identity. Even if you have multiple levels of configuration to retrieve a secret value, the app will use the config sources in the following order:

- first, check the Key Vault.
- if Key Vault not found, check secrets.json file
- finally, check the appsettings.json file.

There are a few areas in your code you need to update, in order to use your Key Vault:

1. Install the nuget packages AppAuthentication and KeyVault NuGet libraries.
  - Microsoft.Azure.Services.AppAuthentication
  - Microsoft.Azure.KeyVault
2. Update Program.cs to configure your application to use Key Vault
3. Inject an IConfiguration object into your controller (MVC) or page model (Razor Pages, shown below)
4. Access specific secrets using the IConfiguration object, e.g. \_configuration["MySecret"]

Below is an example of Program.cs using the WebHostBuilder's **ConfigureAppConfiguration()** method to configure Key Vault. The **keyVaultEndpoint** is the fully-qualified domain name of your Key Vault that you created in Azure.

```
...
using Microsoft.Azure.KeyVault;
using Microsoft.Azure.Services.AppAuthentication;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Configuration.AzureKeyVault; ...
public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((ctx, builder) =>
    {
        var keyVaultEndpoint = GetKeyVaultEndpoint();
```

```

        if (!string.IsNullOrEmpty(keyVaultEndpoint))
        {
            var azureServiceTokenProvider = new
AzureServiceTokenProvider();
            var keyVaultClient = new KeyVaultClient(
                new KeyVaultClient.AuthenticationCallback(
                    azureServiceTokenProvider.KeyVaultTokenCallback));
            builder.AddAzureKeyVault(
                keyVaultEndpoint, keyVaultClient, new
DefaultKeyVaultSecretManager());
        }
    }
).UseStartup<Startup>()
.Build();

```

```
private static string GetKeyVaultEndpoint() =>
"https://<VAULT_NAME>.vault.azure.net/";
```

**NOTE:** Please note that the Web Host Builder in ASP .NET Core 2.x will be replaced by the Generic Host Builder in .NET Core 3.0.

Below is an example of of a Page Model for a Razor Page, e.g. SecretPage.cshtml.cs in the sample app

```

...
using Microsoft.Extensions.Configuration; public class SecretPageModel
: PageModel
{
    public IConfiguration _configuration { get; set; }
    public string Message1 { get; set; }
    public string Message2 { get; set; }
    public string Message3 { get; set; }

    public SecretPageModel(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    public void OnGet()
    {
        Message1 = "My 1st key val = " +
_configuration["MyKeyVaultSecret"];
        Message2 = "My 2nd key val = " +
_configuration["AnotherVaultSecret"];
        Message3 = "My 3nd key val ? " +
_configuration["NonExistentSecret"];
    }
}

```

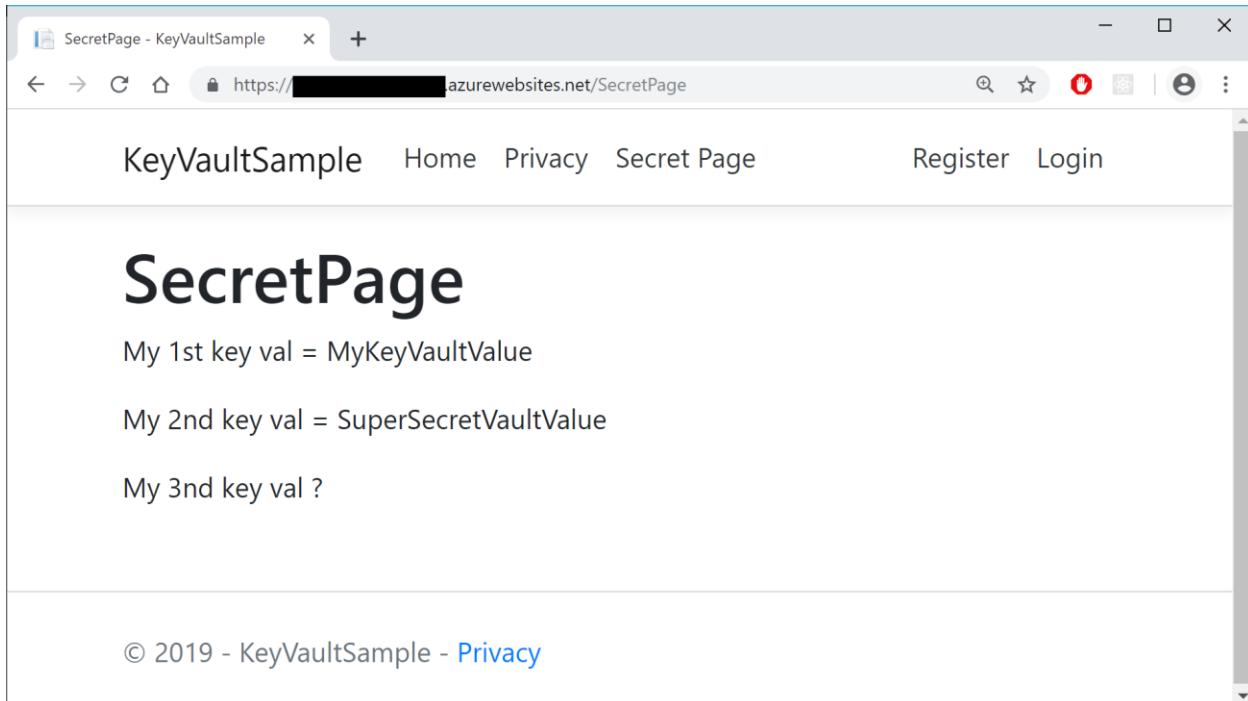
In the above code, the configuration object is injected into the Page Model's **SecretPageModel()**. The values are retrieved from the collection in the **OnGet()** method and assigned to string properties. In the code below, the string properties are accessed from the model class in the corresponding Razor Page, SecretPage.cshtml.

```
@page
@model KeyVaultSample.Pages.SecretPageModel
...
<p>
    @Model.Message1
</p>

<p>
    @Model.Message2
</p>

<p>
    @Model.Message3
</p>
```

Finally, viewing the page allows you to navigate to the Secret Page to view the secret values. Note that I've only created 2 secret values before deploying this instance, so the 3rd value remains blank (*but without generating any errors*).



# Managed Service Identity

There are multiple ways to deploy your ASP .NET Core web app to Azure, including Visual Studio, Azure CLI or a CI/CD pipeline integrated with your source control system. If you need help deploying to Azure App Service, check out the following article from this blog series:

- Deploying ASP .NET Core to Azure App Service: <https://wakeupandcode.com/deploying-asp-net-core-to-azure-app-service/>

You can set up your Managed Service Identity in various ways:

- **Azure Portal:** log in to the Azure Portal and add your app
- **Azure CLI:** use Azure CLI commands to set up MSI
- **Visual Studio:** use the VS IDE while publishing

Once you've created your App Service (even before deploying your Web App to it), you can use the **Azure Portal** to add your app using Managed Service Identity. In the screenshots below, I've added my sample app in addition to my own user access.

- In the **Access Policies** section of the **Key Vault**, you may add one or more access policies.
- In the **Identity** section of the **App Service**, you may update the System-Assigned setting to "On" and make a note of the Object ID, which is defined as a "*Unique identifier assigned to this resource, when it's registered with Azure Active Directory*"

The screenshot shows the 'Access policies' section of an Azure Key Vault. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Keys, Secrets, Certificates, and Access policies (which is highlighted with a red box). The main area displays two entries under 'Click to show advanced access policies': 'Shahed Chowdhuri' (USER) and 'mysamplevaultapp' (APPLICATION). Buttons for Save, Discard, and Refresh are at the top.

KeyVault-AccessPolicies

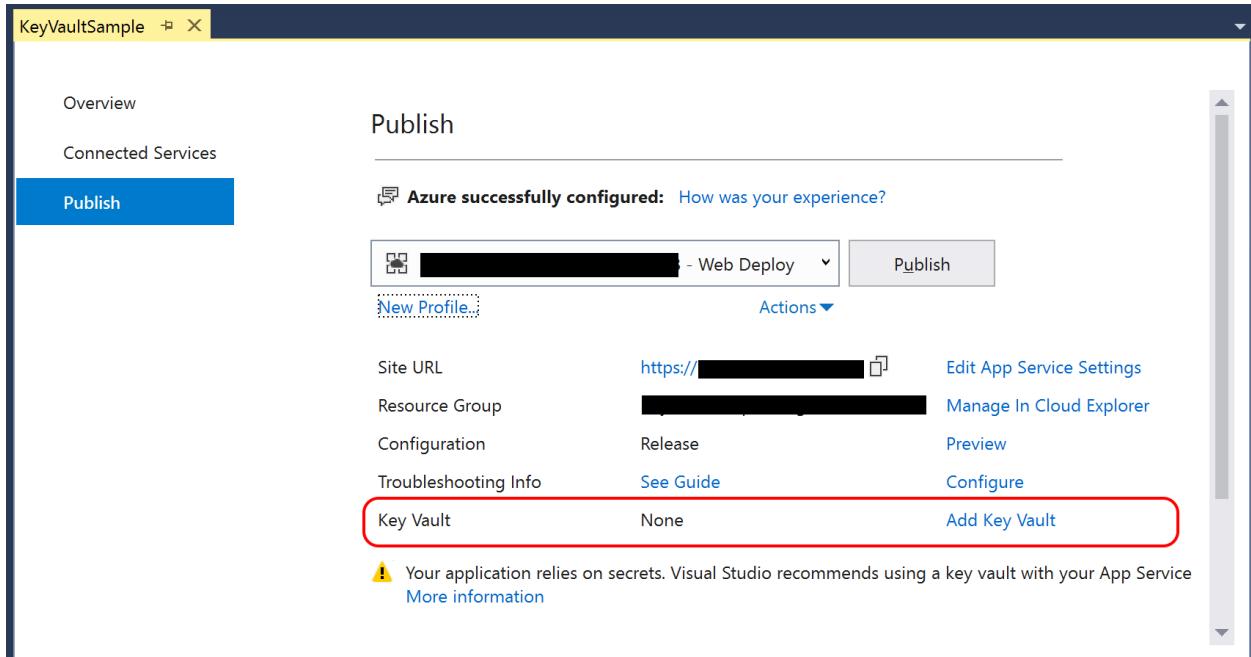
The screenshot shows the 'Identity' settings for an Azure App Service. The left sidebar includes Application settings, Configuration (Preview), Authentication / Authorization, Application Insights, and Identity (which is highlighted with a red box). The main area shows the 'User assigned' tab selected, with a note about system-assigned identities. It includes fields for Status (On), Object ID, and a note about Azure Active Directory integration. Buttons for Save, Discard, and Refresh are at the top.

AppService-Identity

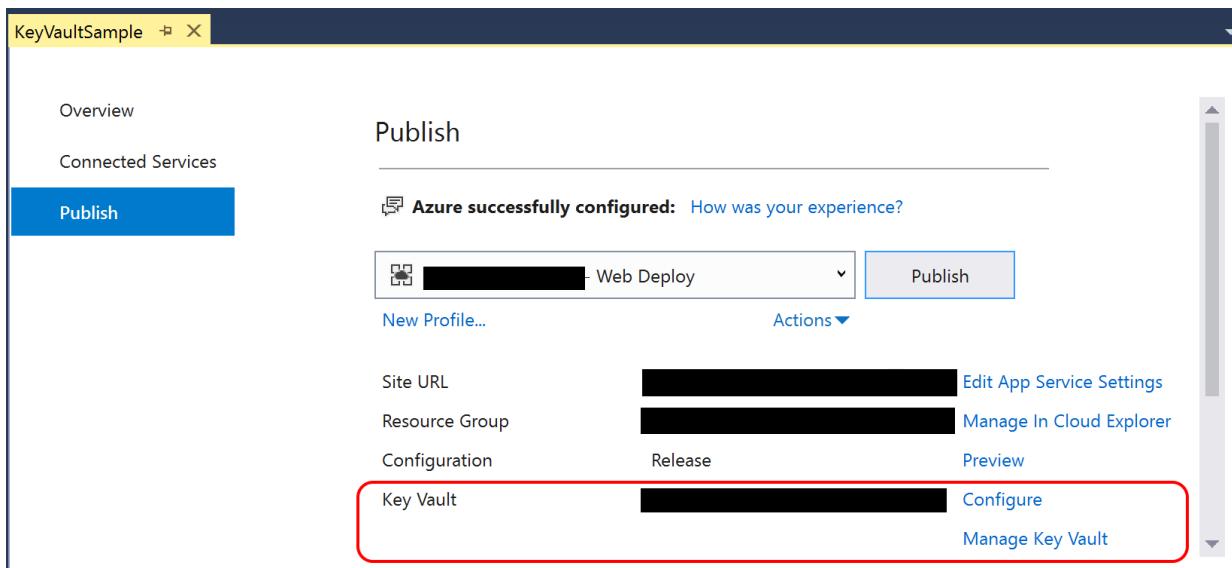
To use the **Azure CLI** to authorize an application to access (or “get”) a key vault, run “**az keyvault set-policy**”, followed by the vault name, the App ID and specific permissions. This is equivalent to enabling the Managed Service Identity for your Web App in the Azure Portal.

```
az keyvault set-policy --name "MyKeyVault" --spn <APP_ID> --secret-permissions get
```

To use **Visual Studio** to use your key vault after deployment, take a look at the Publish screen when deploying via Visual Studio. You’ll notice that there is an option to “Add Key Vault” if it hasn’t been added yet. After you’ve added and enabled Key Vault for your application, the option will change to say “Configure” and “Manage Key Vault”.



Publish (before adding Key Vault)



Publish (Manage Key Vault after adding it)

After adding via Visual Studio during the Publish process, your Publish Profile (*profile* – Web Deploy.pubxml) and Launch Settings profiles (launchSettings.json) should contain the fully qualified domain name for your Key Vault in Azure. You should not include these files in your source control system.

## References

- Azure Key Vault Developer's Guide: <https://docs.microsoft.com/en-us/azure/key-vault/key-vault-developers-guide>
- Add Key Vault support to your ASP.NET project using Visual Studio: <https://docs.microsoft.com/en-us/azure/key-vault/vs-key-vault-add-connected-service>
- Key Vault Configuration Provider Sample App: <https://github.com/aspnet/Docs/tree/master/aspnetcore/security/key-vault-configuration/sample>
- Getting Started with Azure Key Vault with .NET Core: <https://azure.microsoft.com/en-us/resources/samples/key-vault-dotnet-core-quickstart/>
- Azure-Samples/key-vault-dotnet-core-quickstart: <https://github.com/Azure-Samples/key-vault-dotnet-core-quickstart>

- Using Azure Key Vault with ASP.NET Core and Azure App Services: <https://damienbod.com/2018/12/23/using-azure-key-vault-with-asp-net-core-and-azure-app-services/>
- Azure Key Vault configuration provider in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/security/key-vault-configuration>
- slide-decks/Protecting App Secrets.pptx: <https://github.com/scottaddie/slides/blob/master/Protecting%20App%20Secrets.pptx>
- Set and retrieve a secret from Azure Key Vault by using a node web app – Azure Key Vault: <https://docs.microsoft.com/en-us/azure/key-vault/quick-create-net>
- Azure Key Vault managed storage account – CLI: <https://docs.microsoft.com/en-us/azure/key-vault/key-vault-ovw-storage-keys>
- Service-to-service authentication to Azure Key Vault using .NET: <https://docs.microsoft.com/en-us/azure/key-vault/service-to-service-authentication>
- Managed identities for Azure resources: <https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/overview>

# Logging in ASP .NET Core

By Shahed C on March 25, 2019

5 Replies

This is the **twelfth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

- L is for Logging in ASP .NET Core
- Log Messages
- Logging Providers
- JSON Configuration
- Log Categories
- Exceptions in Logs
- Structured Logging with Serilog
- References

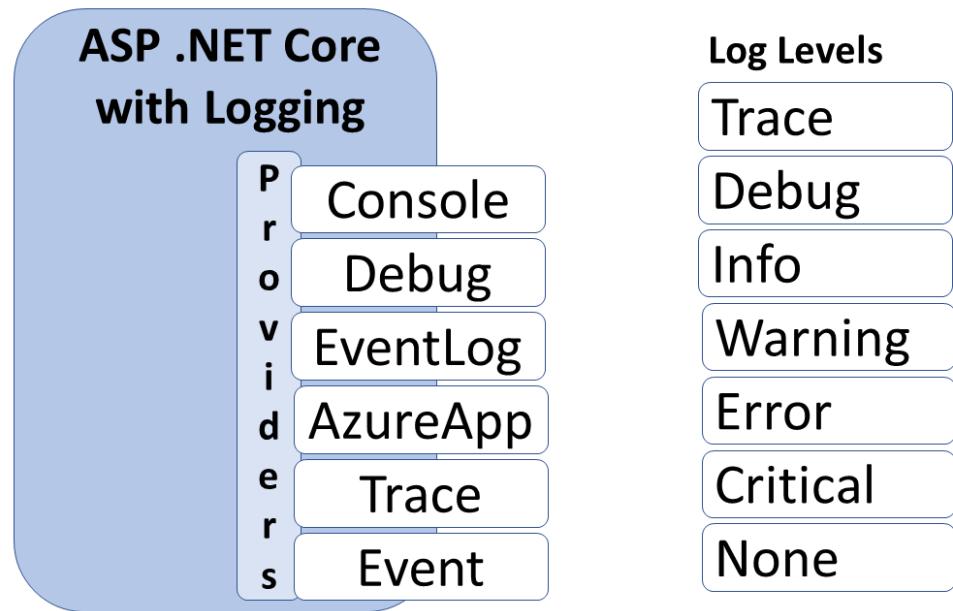
# L is for Logging in ASP .NET Core

You *could* write a fully functional ASP .NET Core web application without any logging. But in the real world, you *should* use some form of logging. This blog post provides an overview of how you can use the *built-in* logging functionality in ASP .NET Core web apps. While we won't go deep into 3rd-party logging solutions such as Serilog in this article (*stay tuned for a future article on that topic*), you should definitely consider a robust semantic/structured logging solution for your projects.

To follow along, take a look at the sample project on Github:



Logging Sample: <https://github.com/shahedc/AspNetCoreLoggingSample>



Logging in ASP .NET Core

## Log Messages

The simplest log message includes a call to the extension method **ILogger.Log()** by passing on a **LogLevel** and a text string. Instead of passing in a LogLevel, you could also call a specific Log method such as **LogInformation()** for a specific LogLevel. Both examples are shown below:

```
// Log() method with LogLevel passed in  
_logger.Log(LogLevel.Information, "some text");  
  
// Specific LogXX() method, e.g. LogInformation()  
_logger.LogInformation("some text");
```

LogLevel values include Trace, Debug, Information, Warning, Error, Critical and None. These are all available from the namespace **Microsoft.Extensions.Logging**. For a more structured logging experience, you should also pass in meaningful variables/objects following the templated message string, as all the Log methods take in a set of parameters defined as “params object[] args”.

```
public static void Log (  
    this ILogger logger, LogLevel logLevel, string message, params  
    object[] args);
```

This allows you to pass those values to specific logging providers, along with the message itself. It's up to each logging provider on how those values are captured/stored, which you can also configure further. You can then query your log store for specific entries by searching for those arguments. In your code, this could look something like this:

```
_logger.LogInformation("some text for id: {someUsefulId}",  
someUsefulId);
```

Even better, you can add your own **EventId** for each log entry. You can facilitate this by defining your own set of integers, and then passing an int value to represent an EventId. The EventId type is a struct that includes an implicit operator, so it essentially calls its own constructor with whatever int value you provide.

```
_logger.LogInformation(someEventId, "some text for id:  
{someUsefulId}", someUsefulId);
```

In the sample project, we can see the use of a specific integer value for each EventId, as shown below:

```
// Step X: kick off something here  
_logger.LogInformation(LoggingEvents.Step1KickedOff, "Step {stepId}  
Kicked Off.", stepId);  
  
// Step X: continue processing here  
_logger.LogInformation(LoggingEvents.Step1InProcess, "Step {stepId} in  
process...", stepId);  
  
// Step X: wrap it up  
_logger.LogInformation(LoggingEvents.Step1Completed, "Step {stepId}
```

```
completed!", stepId);
```

The integer values can be whatever you want them to be. An example is shown below:

```
public class LoggingEvents
{
    public const int ProcessStarted = 1000;

    public const int Step1KickedOff = 1001;
    public const int Step1InProcess = 1002;
    public const int Step1Completed = 1003;

    ...
}
```

## Logging Providers

The default template-generated web apps include a call to **CreateDefaultBuilder()** in Program.cs, which automatically includes adds the Console and Debug providers. As of ASP.NET Core 2.2, the EventSource provider is also automatically added by the default builder.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
```

**NOTE:** As mentioned in an earlier post in this blog series, the Web Host Builder will be replaced by the Generic Host Builder with the release of .NET Core 3.0.

If you wish to add your own set of logging providers, you can expand the call to CreateDefaultBuilder(), clear the default providers, and then add your own. The built-in providers now include **Console**, **Debug**, **EventLog**, **AzureAppServices**, **TraceSource** and **EventSource**.

```
public static IWebHostBuilder CreateWebHostBuilder(
    string[] args) => WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging(logging =>
    {
        // clear default logging providers
        logging.ClearProviders();

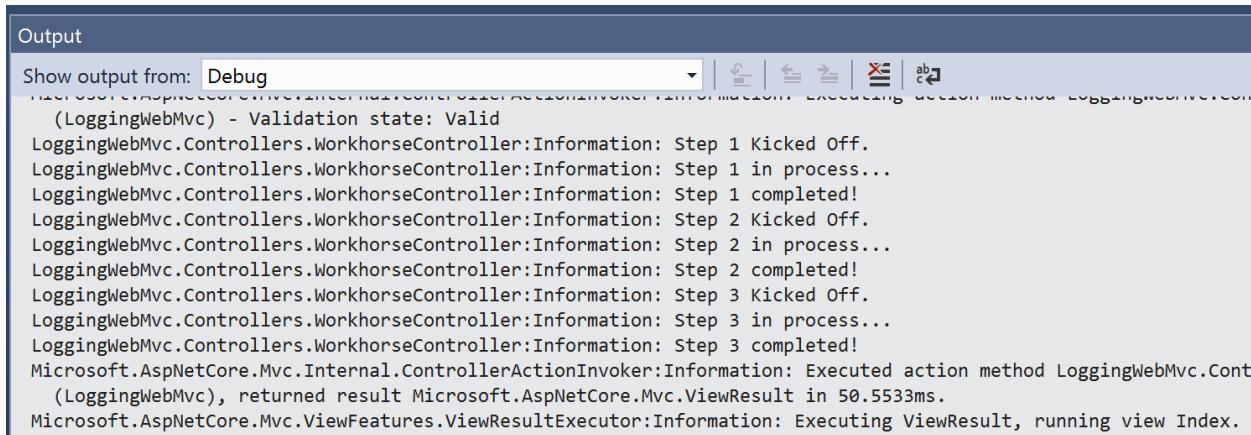
        // add built-in providers manually, as needed
        logging.AddConsole();
        logging.AddDebug();
        logging.AddEventLog();
```

```

        logging.AddEventSourceLogger();
        logging.AddTraceSource(sourceSwitchName);
    });
}

```

The screenshots below show the log results viewable in Visual Studio's Debug Window and in the Windows 10 Event Viewer. Note that the EventId's integer values (that we had defined) are stored in the EventId field as numeric value in the Windows Event Viewer log entries.



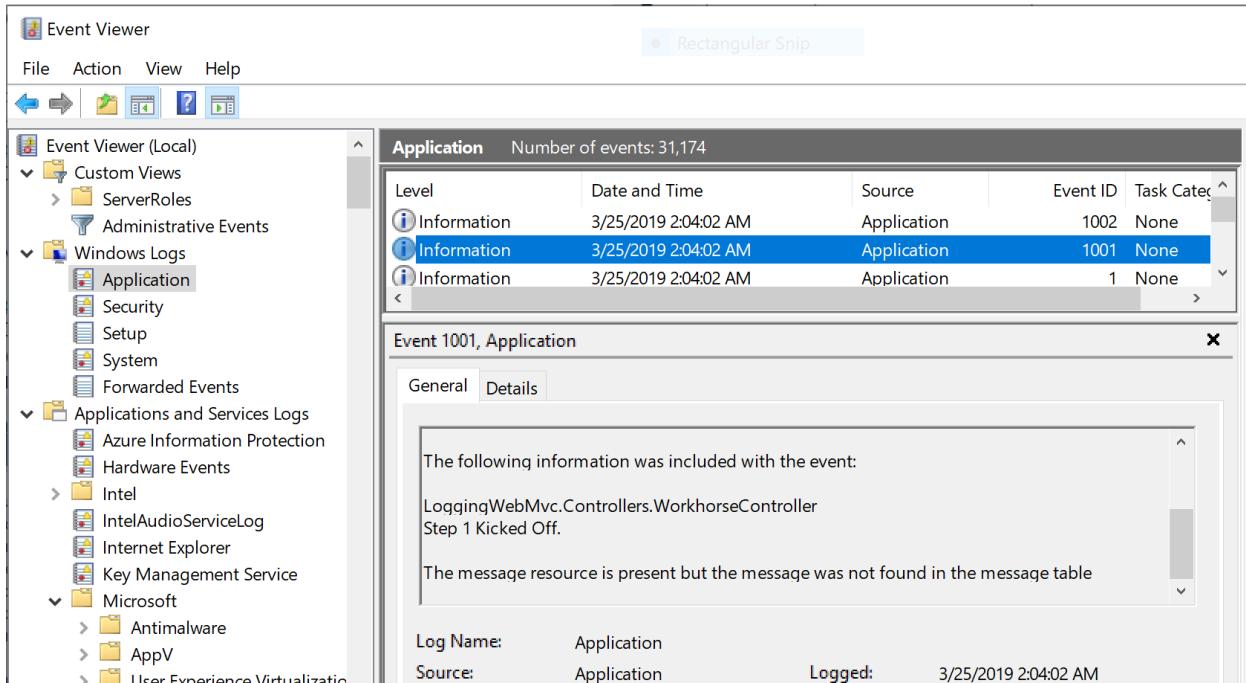
The screenshot shows the Visual Studio Output window with the title "Output". A dropdown menu at the top says "Show output from: Debug". Below the dropdown, there is a scrollable list of log messages:

```

Microsoft.AspNetCore.Mvc.Internal.ControllerInvoker:Information: Executing action method LoggingWebMvc.Controllers.WorkhorseController.Index()
(LoggingWebMvc) - Validation state: Valid
LoggingWebMvc.Controllers.WorkhorseController:Information: Step 1 Kicked Off.
LoggingWebMvc.Controllers.WorkhorseController:Information: Step 1 in process...
LoggingWebMvc.Controllers.WorkhorseController:Information: Step 1 completed!
LoggingWebMvc.Controllers.WorkhorseController:Information: Step 2 Kicked Off.
LoggingWebMvc.Controllers.WorkhorseController:Information: Step 2 in process...
LoggingWebMvc.Controllers.WorkhorseController:Information: Step 2 completed!
LoggingWebMvc.Controllers.WorkhorseController:Information: Step 3 Kicked Off.
LoggingWebMvc.Controllers.WorkhorseController:Information: Step 3 in process...
LoggingWebMvc.Controllers.WorkhorseController:Information: Step 3 completed!
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executed action method LoggingWebMvc.Controllers.WorkhorseController.Index(), returned result Microsoft.AspNetCore.Mvc.ViewResult in 50.5533ms.
Microsoft.AspNetCore.Mvc.ViewFeatures.ViewResultExecutor:Information: Executing ViewResult, running view Index.

```

Logging: VS2017 Debug Window



The screenshot shows the Windows Event Viewer interface. The left pane shows a tree view of logs: Event Viewer (Local), Custom Views, Server Roles, Administrative Events, Windows Logs (selected), Applications and Services Logs, and Microsoft. The right pane shows the "Application" log with 31,174 events. A specific event is selected, showing its details in the bottom pane:

**Event 1001, Application**

Level	Date and Time	Source	Event ID	Task Category
Information	3/25/2019 2:04:02 AM	Application	1002	None
Information	3/25/2019 2:04:02 AM	Application	1001	None
Information	3/25/2019 2:04:02 AM	Application	1	None

**Event 1001, Application**

The following information was included with the event:  
**LoggingWebMvc.Controllers.WorkhorseController**  
Step 1 Kicked Off.

The message resource is present but the message was not found in the message table

Log Name: Application  
Source: Application  
Logged: 3/25/2019 2:04:02 AM

Logging; Windows Event Viewer

For the *Event Log provider*, you'll also have to add the following NuGet package and corresponding using statement:

```
Microsoft.Extensions.Logging.EventLog
```

For the *Trace Source provider*, a “source switch” can be used to determine if a trace should be propagated or ignored. For more information on the Trace Source provider and the Source Switch it uses check out the official docs at:

- *SourceSwitch Class (System.Diagnostics)*: <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.sourceswitch>

For more information on adding logging providers and further customization, check out the official docs at:

- *Add Providers section of Logging in ASP.NET Core*: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging#add-providers>

**But wait a minute**, what happened to the Azure App Services provider mentioned earlier? Why isn't there a call to add it as a logging provider? Well, you should be aware that there is a method for adding Azure Web App Diagnostics as a logging provider:

```
logging.AddAzureWebAppDiagnostics();
```

However, you would only have to call this **AddAzureWebAppDiagnostics()** method if you're targeting .NET Framework. You shouldn't call it if targeting .NET Core. With .NET Core 3.0, ASP.NET Core will run *only* on .NET Core so you don't have to worry about this at all. When you deploy the web app to Azure App Service, this logging provider is automatically available for your use. (*We will cover this in more detail in a future blog post.*)

## JSON Configuration

One way to configure each Logging Provider is to use your `appsettings.json` file. Depending on your environment, you could start with `appsettings.Development.json` or App Secrets in development, and then use environment variables, Azure Key Vault in other environments. You may refer to earlier blog posts from 2018 and 2019 for more information on the following:

- Your Web App Secrets in ASP .NET Core: <https://wakeupandcode.com/your-web-app-secrets-in-asp-net-core/>
- Key Vault for ASP .NET Core Web Apps: <https://wakeupandcode.com/key-vault-for-asp-net-core-web-apps/>

In your local JSON config file, your configuration uses the following syntax:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "Category1": "Information",
      "Category2": "Warning"
    },
    "SpecificProvider": {
      "ProviderProperty": true
    }
  }
}
```

The configuration for **LogLevel** sets one or more categories, including the **Default** category when no category is specified. Additional categories (e.g. System, Microsoft or any custom category) may be set to one of the aforementioned LogLevel values.

The **LogLevel** block can be followed by one or more provider-specific blocks (e.g. **Console**) to set its properties, e.g. **IncludeScopes**. Such an example is shown below.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    },
    "Console": {
      "IncludeScopes": true
    }
  }
}
```

To set logging filters in code, you can use the **AddFilter ()** method for specific providers or all providers in your Program.cs file. The following syntax can be used to add filters for your logs.

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging =>
        logging.AddFilter("Category1", LogLevel.Level1)
        .AddFilter<SomeProvider>("Category2", LogLevel.Level2));
```

In the above sample, the following placeholders can be replaced with:

- **CategoryX**: System, Microsoft, custom categories
- **LogLevel.LevelX**: Trace, Debug, Information, Warning, Error, Critical, None
- **SomeProvider**: Debug, Console, other providers

## Log Categories

To set a category when logging an entry, you may set the string value when creating a logger. If you don't set a value explicitly, the fully-qualified namespace + class name is used. In the WorkhorseController class seen in your example, the log results seen in the Debug window and Event Viewer were seen to be:

```
LoggingWebMvc.Controllers.WorkhorseController
```

This is the *category name* created using the controller class name passed to the constructor in WorkHoseController as shown below:

```
private readonly ILogger _logger;

public WorkhorseController(ILogger<WorkhorseController> logger)
{
    _logger = logger;
}
```

If you wanted to set this value yourself, you could change the code to the following:

```
private readonly ILogger _logger;

public WorkhorseController	ILoggerFactory logger)
{
    _logger =
logger.CreateLogger("LoggingWebMvc.Controllers.WorkhorseController");
}
```

The end results will be the same. However, you may notice that there are a couple of differences here:

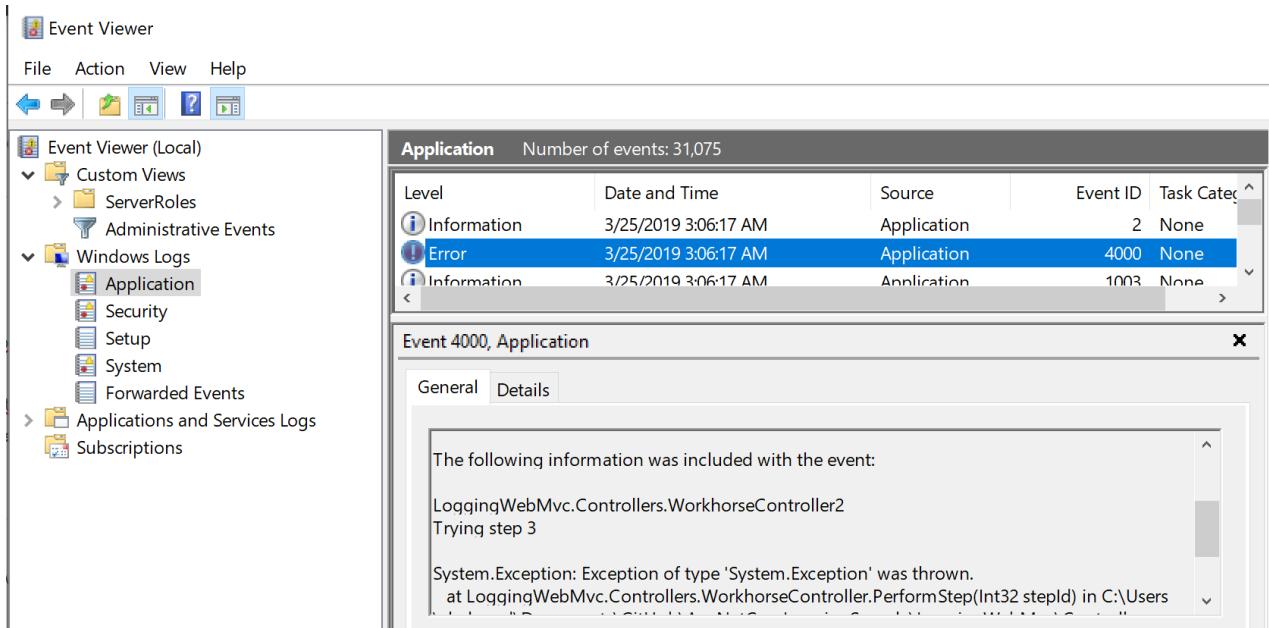
1. Instead of **ILogger<classname>** we are now passing in an **ILoggerFactory** type as the logger.
2. Instead of just assigning the injected logger to the private **\_logger** variable, we are now calling the factory method **CreateLogger()** with the desired string value to set the category name.

## Exceptions in Logs

In addition to EventId values and Category Names, you may also capture Exception information in your application logs. The various Log extensions provide an easy way to pass an exception by passing the Exception object itself.

```
try
{
    // try something here
    throw new Exception();
} catch (Exception someException)
{
    _logger.LogError(eventId, someException, "Trying step {stepId}",
stepId);
    // continue handling exception
}
```

Checking the Event Viewer, we may see a message as shown below. The **LogLevel** is shown as “Error” because we used the **LogError()** extension method in the above code, which is forcing an Exception to be thrown. The details of the Exception is displayed in the log as well.



Logging; Exception in Event Viewer

## Structured Logging with Serilog

At the very beginning, I mentioned the possibilities of structured logging with 3rd-party providers. There are many solutions that work with ASP .NET Core, including (but not limited to) elmah, NLog and Serilog. Here, we will take a brief look at Serilog.

Similar to the built-in logging provider described throughout this article, you should include variables to assign template properties in all log messages, e.g.

```
Log.Information("This is a message for {someVariable}", someVariable);
```

To make use of Serilog, you'll have to perform the following steps:

1. grab the appropriate NuGet packages: Serilog, Hosting, various Sinks, e,g, Console
2. use the Serilog namespace, e.g. **using Serilog**

3. create a new `LoggerConfiguration()` in your `Main()` method
4. call `UseSerilog()` when creating your Host Builder
5. write log entries using methods from the `Log` static class.

For more information on Serilog, check out the following resources:

- Getting Started: <https://github.com/serilog/serilog/wiki/Getting-Started>
- Writing Log Events: <https://github.com/serilog/serilog/wiki/Writing-Log-Events>
- Serilog Tutorial for .NET Logging: <https://stackify.com/serilog-tutorial-net-logging/>
- Adding Serilog to the ASP.NET Core Generic Host: <https://andrewlock.net/adding-serilog-to-the-asp-net-core-generic-host/>
- Asp.Net Core 2 Logging With Serilog: <http://hamidmosalla.com/2018/02/15/asp-net-core-2-logging-with-serilog-and-microsoft-sql-server-sink/>

## References

- Logging in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/>
- ASP.NET Core Logging with Azure App Service and Serilog: <https://devblogs.microsoft.com/aspnet/asp-net-core-logging/>
- Creating a rolling file logging provider for ASP.NET Core 2.0: <https://andrewlock.net/creating-a-rolling-file-logging-provider-for-asp-net-core-2-0/amp/>
- Explore .NET trace logs in Azure Application Insights with `ILogger`: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/logger>
- Azure Application Insights for ASP.NET Core: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/asp-net-core>
- Don't let ASP.NET Core Console Logging Slow your App down: <https://weblog.west-wind.com/posts/2018/Dec/31/Dont-let-ASPNET-Core-Default-Console-Logging-Slow-your-App-down>

# Middleware in ASP .NET Core

By Shahed C on April 3, 2019

3 Replies

This is the **thirteenth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**

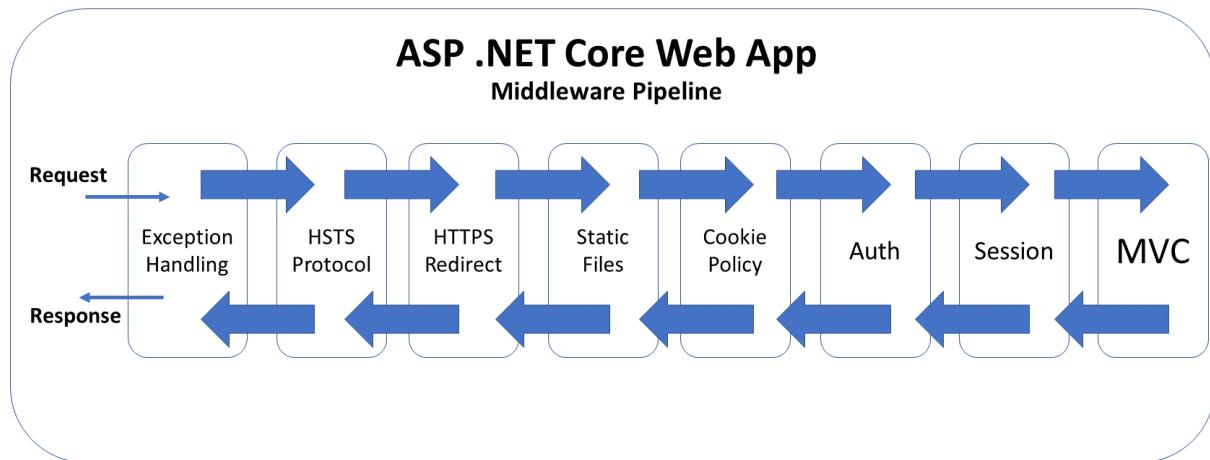


## In this Article:

- M is for Middleware in ASP .NET Core
- How It Works
- Built-in Middleware
- Branching out with Run, Map & Use
- References

## M is for Middleware in ASP .NET Core

If you've been following my blog series (or if you've done any work with ASP .NET Core at all), you've already worked with the Middleware pipeline. When you create a new project using one of the built-in templates, your project is already supplied with a few calls to add/configure middleware services and then use them. This is accomplished by adding the calls to the `Startup.cs` configuration methods.



The above diagram illustrates the typical order of middleware layers in an ASP .NET Core web application. The order is very important, so it is necessary to understand the placement of each *request delegate* in the pipeline.

To follow along, take a look at the sample project on Github:



Middleware Sample: <https://github.com/shahedc/AspNetCoreMiddlewareSample>

## How It Works

When an HTTP request comes in, the first request delegate handles that request. It can either pass the request down to the next in line or short-circuit the pipeline by preventing the request from propagating further. This is very useful across multiple scenarios, e.g. serving static files without the need for authentication, handling exceptions before anything else, etc.

The returned response travels back in the reverse direction back through the pipeline. This allows each component to run code both times: when the request arrives and also when the response is on its way out.

Here's what the **Configure()** method looks like, in a template-generated Startup.cs file:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorResponse();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

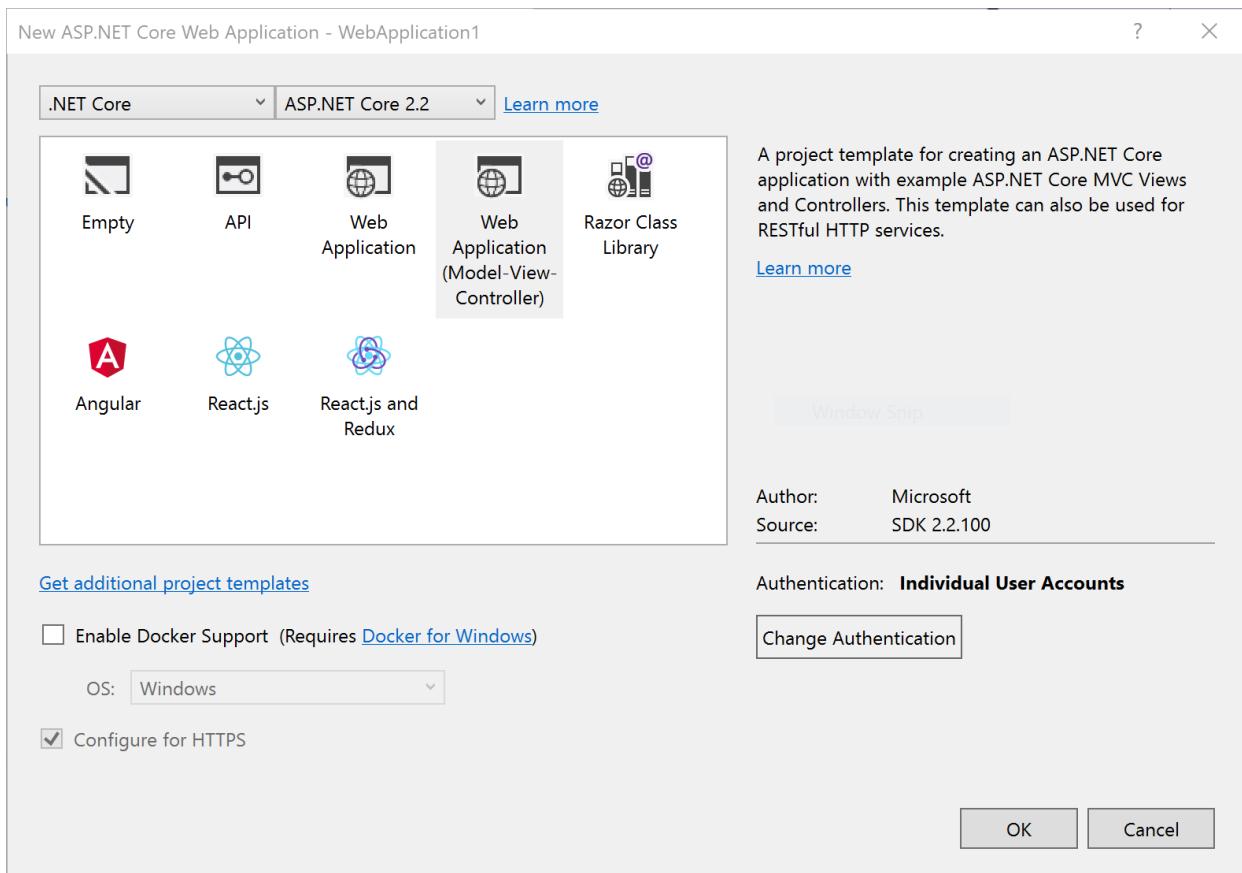
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseAuthentication();

    app.UseSession();

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

I have added a call to **UseSession()** to call the Session Middleware, as it wasn't included in the template-generated project. I also enabled authentication and HTTPS when creating the template.



In order to configure the use of the Session middleware, I also had to add the following code in my **ConfigureServices()** method:

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddDistributedMemoryCache();
    services.AddSession(options =>
    {
        options.IdleTimeout = TimeSpan.FromSeconds(10);
        options.Cookie.HttpOnly = true;
        options.Cookie.IsEssential = true;
    });
}

services.AddMvc();
```

```
}
```

The calls to **AddDistributedMemoryCache()** and **AddSession()** ensure that we have enabled a (memory cache) backing store for the session and then prepared the Session middleware for use. In the Razor Pages version of Startup.cs, the **Configure()** method is a little simpler, as it doesn't need to set up MVC routes for controllers in the call to **useMvc()**.

## Built-In Middleware

The information below explains how the built-in middleware works, and why the order is important. The **UseXYZ()** methods are merely extension methods that are prefixed with the word "Use" as a useful convention, making it easy to discover Middleware components when typing code. Keep this in mind when developing custom middleware.

### *Exception Handling:*

- **UseDeveloperExceptionPage()** & **UseDatabaseErrorResponse()**: used in *development* to catch run-time exceptions
- **UseExceptionHandler()**: used in *production* for run-time exceptions

Calling these methods first ensures that exceptions are caught in any of the middleware components that follow. For more information, check out the detailed post on Handling Errors in ASP .NET Core, earlier in this series.

### *HSTS & HTTPS Redirection:*

- **UseHsts()**: used in production to enable HSTS (HTTP Strict Transport Security Protocol) and enforce HTTPS.
- **UseHttpsRedirection()**: forces HTTP calls to automatically redirect to equivalent HTTPS addresses.

Calling these methods next ensure that HTTPS can be enforced before resources are served from a web browser. For more information, check out the detailed post on Protocols in ASP .NET Core: HTTPS and HTTP/2.

#### ***Static Files:***

- **UseStaticFiles():** used to enable static files, such as HTML, JavaScript, CSS and graphics files. Called early on to avoid the need for authentication, session or MVC middleware.

Calling this before authentication ensures that static files can be served quickly without unnecessarily triggering authentication middleware. For more information, check out the detailed post on JavaScript, CSS, HTML & Other Static Files in ASP .NET Core.

#### ***Cookie Policy:***

- **UseCookiePolicy():** used to enforce cookie policy and display GDPR-friendly messaging

Calling this before the next set of middleware ensures that the calls that follow can make use of cookies if consented. For more information, check out the detailed post on Cookies and Consent in ASP .NET Core.

#### ***Authentication, Authorization & Sessions:***

- **UseAuthentication():** used to enable authentication and then subsequently allow authorization.
- **UseSession():** manually added to the Startup file to enable the Session middleware.

Calling these after cookie authentication (but before the MVC middleware) ensures that cookies can be issued as necessary and that the user can be authenticated before the MVC engine kicks in. For more information, check out the detailed post on Authentication & Authorization in ASP .NET Core.

#### ***MVC & Routing:***

- **UseMvc()**: enables the use of MVC in your web application, with the ability to customize routes for your MVC application and set other options.
- **routes.MapRoute()**: set the default route and any custom routes when using MVC.

To create your own *custom* middleware, check out the official docs at:

- Write custom ASP.NET Core middleware: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/write>

## Branching out with Run, Map & Use

The so-called “request delegates” are made possible with the implementation of three types of extension methods:

- Run: enables short-circuiting with a terminal middleware delegate
- Map: allows branching of the pipeline path
- Use: call the next desired middleware delegate

If you’ve tried out the absolute basic example of an ASP .NET Core application (generated from the Empty template), you may have seen the following syntax in your Startup.cs file. Here, the **app.Run()** method terminates the pipeline, so you wouldn’t add any additional middleware calls after it.

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

The call to **app.Use()** can be used to trigger the next middleware component with a call to **next.Invoke()** as shown below:

```

public void Configure(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        // ...
        await next.Invoke();
        // ...
    });
}

app.Run(...);
}

```

If there is no call to **next.Invoke()**, then it essentially short-circuits the middleware pipeline. Finally, there is the **Map()** method, which creates separate forked paths/branches for your middleware pipeline and multiple terminating ends. The code snippet below shows the contents of a Startup.cs file with mapped branches:

```

// first branch handler
private static void HandleBranch1(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Branch 1.");
    });
}

// second branch handler
private static void HandleBranch2(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Branch 2.");
    });
}

public void Configure(IApplicationBuilder app)
{
    app.Map("/mappedBranch1", HandleBranch1);

    app.Map("/mappedBranch2", HandleBranch2);

    // default terminating Run() method
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Terminating Run()
method.");
    });
}

```

```
}
```

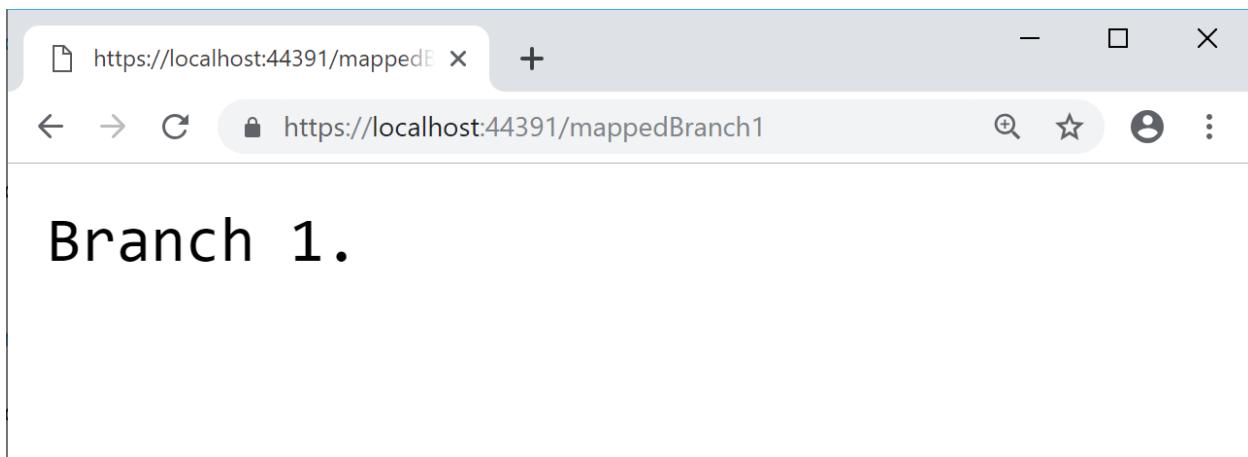
In the **Configure()** method, each call to **app.Map()** establishes a separate branch that can be triggered with the appropriate relative path. Each handler method then contains its own terminating **map.Run()** method.

For example:

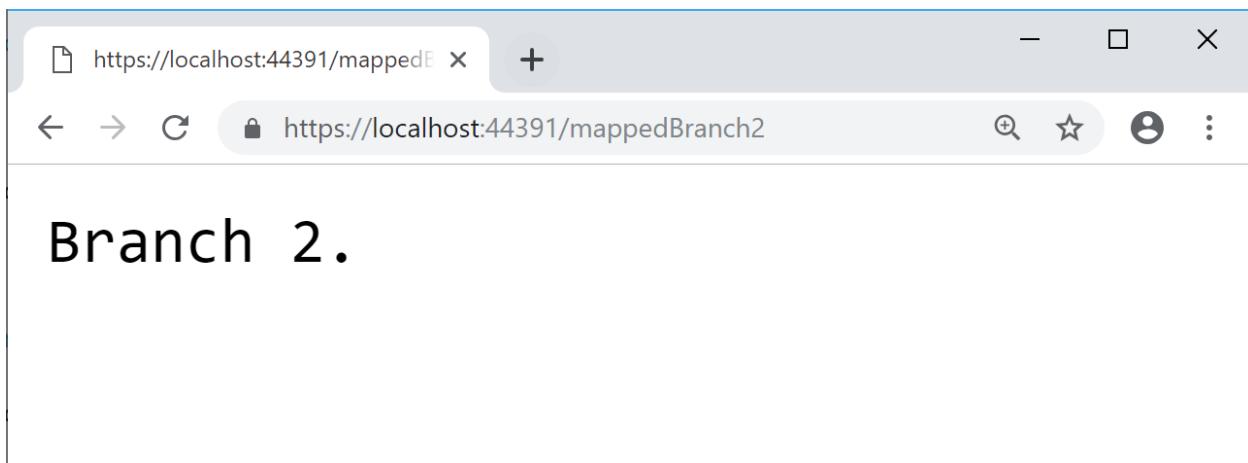
- accessing */mappedBranch1* in an HTTP request will call **HandleBranch1()**
- accessing */mappedBranch2* in an HTTP request will call **HandleBranch2()**
- accessing the / root of the web application will call the default terminating **Run()** method
- specifying an invalid path will also call the default **Run()** method

The screenshots below illustrate the results of the above scenarios:

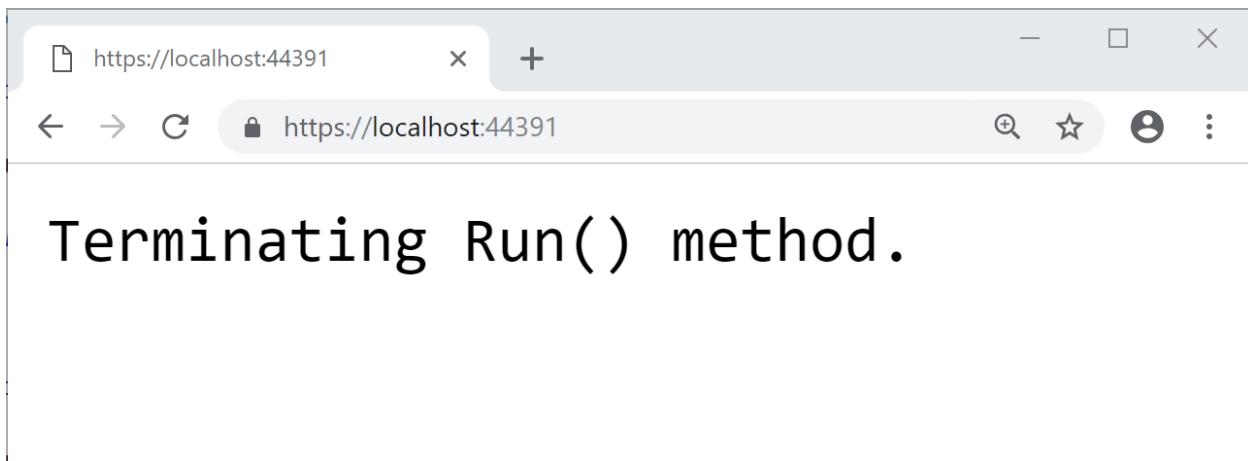
*Scenario 1: https://localhost:44391/mappedBranch1*



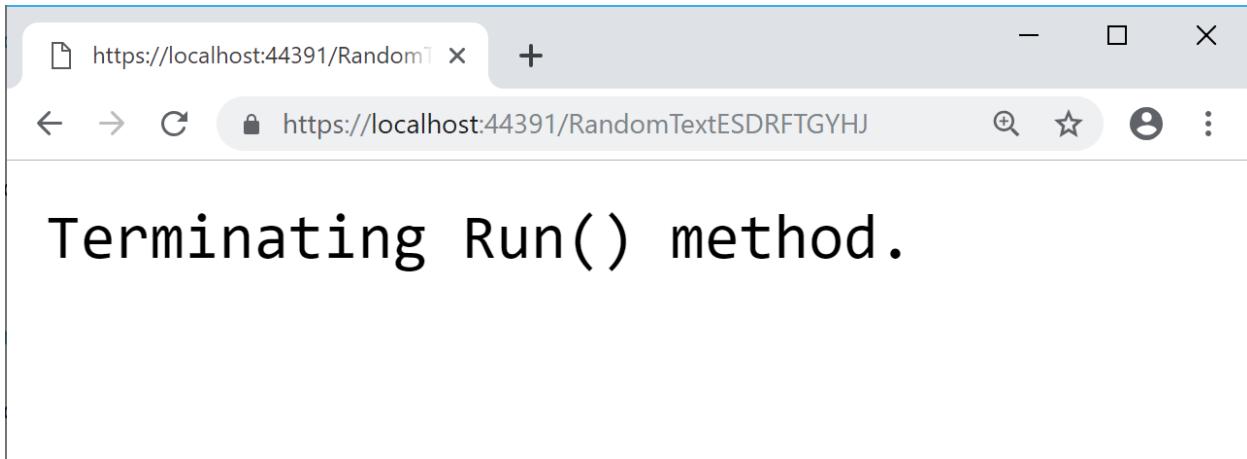
*Scenario 2: https://localhost:44391/mappedBranch2*



*Scenario 3:* <https://localhost:44391/>



*Scenario 4:* <https://localhost:44391/RandomTextESDRFTGYHJ>



## References

- ASP.NET Core Middleware: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware>
- Write custom ASP.NET Core middleware: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/write>
- Session and app state in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/app-state>

Hope you enjoyed learning about Middleware in ASP.NET Core. Here is some feedback I received via Twitter: *"Learned so [much] about #ASPNetCore middleware from this article. The templates should have comments in them with this stuff!"*

- Tweet: <https://twitter.com/SturlaThorvalds/status/1114802739744321536>

— Sturla Þorvaldsson (@SturlaThorvalds) April 7, 2019

# .NET Core 3.0, VS2019 and C# 8.0 for ASP .NET Core developers

By Shahed C on April 8, 2019

9 Replies

This is the **fourteenth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

- N is for .NET Core 3.0, VS2019 and C# 8.0
- Visual Studio 2019
- .NET Core 3.0
- C# 8.0 features
- ASP .NET Core 3.0
- References

# N is for .NET Core 3.0, VS2019 and C# 8.0

After completing 13 blog posts in this A-Z series, Visual Studio 2019 has now been released. While this is just a coincidence, this is a great opportunity to focus on .NET Core 3.0, VS 2019 and C# 8.0 language features for ASP .NET Core developers. This blog post provides an overview of everything you need to know to get started with the above, as an ASP .NET Core developer.



## Visual Studio 2019

First things first: now that Visual Studio 2019 has been released, where can you download it from? Start with the main download page and then select the edition you need:

- VS2019 Downloads: <https://visualstudio.microsoft.com/downloads/>

- Community Edition
- Professional Edition
- Enterprise Edition

As before, the Community Edition (comparable to Pro) is free for students, open-source contributors and individuals. The Pro and Enterprise editions add additional products and services from small teams to enterprise companies.

*But wait!* What if you can't stay online for the length of the installation or need to reinstall quickly at a later date? If you need an offline installer, check out the instructions on the following page:

- Create an offline installation: <https://docs.microsoft.com/en-us/visualstudio/install/create-an-offline-installation-of-visual-studio?view=vs-2019>

What are some cool new and improved features to be aware of? There are so many that I stitched together a series of tweets from Amanda Silver (Director of Program Management for Dev Tools at Microsoft) and created the following thread:

- Twitter thread: <https://twitter.com/shahedC/status/1113177299652837376>

So many cool things in #VS2019! Here's what's awesome and new and improved!

Tweets from @amandaksilver 

— Shahed (on Leave) (@shahedC) April 2, 2019

The aforementioned thread highlights the following features. Click each hyperlink in the list below for more info on each.

- Live Share: Available as an extension in VS Code, Live Share is installed by default with VS2019. Easily collaborate with other developers while coding in real-time!

- Intellisense: Use AI to write better code. Choose to share what you want with others or keep things private.
- Git-first workflows: Choose to create a new project from a source code repo or use a template. The new start window provides more options up front.
- Debug search: Search while debugging. Type in search filters in the Watch, Locals, and Autos panels.
- Snapshot debugging: Available in the Enterprise Edition, snapshot debugging allows you to get a snapshot of your app's execution after deployment. This includes cloud deployments, Azure VMs and Kubernetes containers.
- VS Search: Dynamic search results include commands, menus, components and templates. Note that this was formerly known as *Quick Launch*.
- App Service Debugging: Attach the debugger to your app running in Azure App Service!
- App Service Connectivity: Connect your web app to Azure App Service with ease, including App Insights monitoring.
- Azure Monitor: Use Azure Monitor to get additional insight on your deployed app!

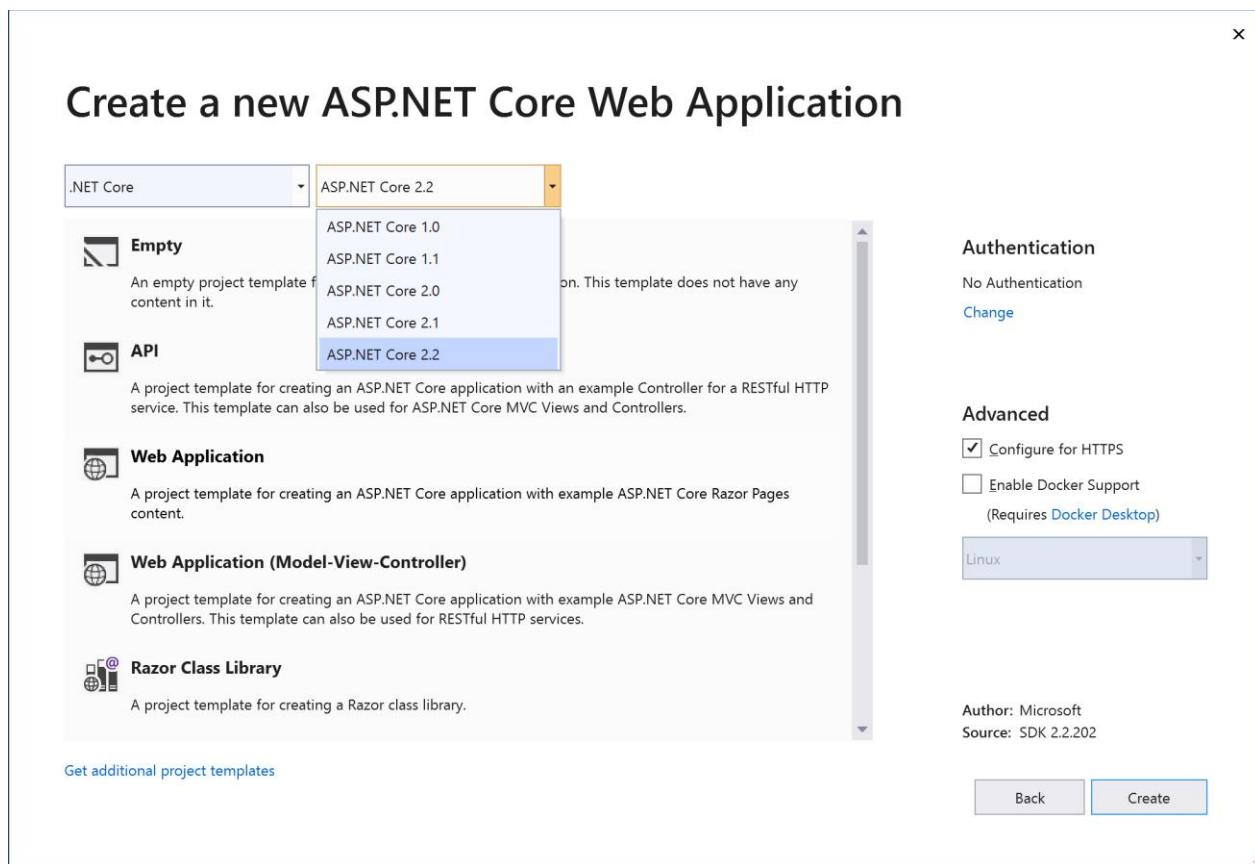
If you prefer to sit back and relax and just watch new product announcements, I put together a handy list of YouTube videos from the VS2019 launch event. This playlist kicks off with the 50-minute keynote, is followed by a string of videos and ends with a cumulative 7-hour video if you prefer to watch all at once.

- VS2019 Launch event  
playlist: <https://www.youtube.com/watch?v=DANLUUIUrcM&list=PLPmfQOWse5Gr4pMcYFmwHW-hDUWmbT6B>

## .NET Core 3.0

If you've downloaded Visual Studio 2019, you may have followed the following steps:

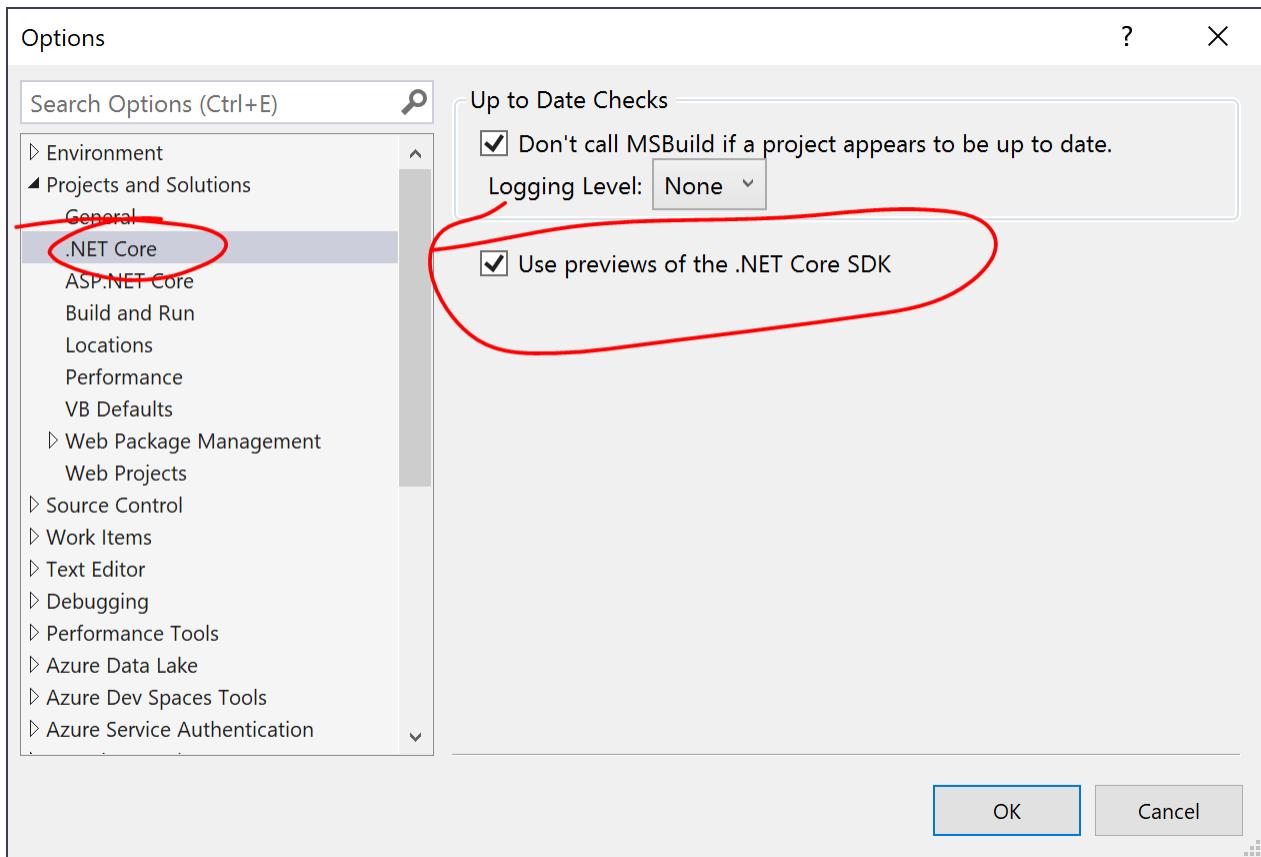
1. Download Visual Studio 2019
2. Click **File | New | Project** (or create new from splash screen)
3. Create a new ASP .NET Core Web App
4. Select .NET Core 3.0 as the project type/platform... *right?*



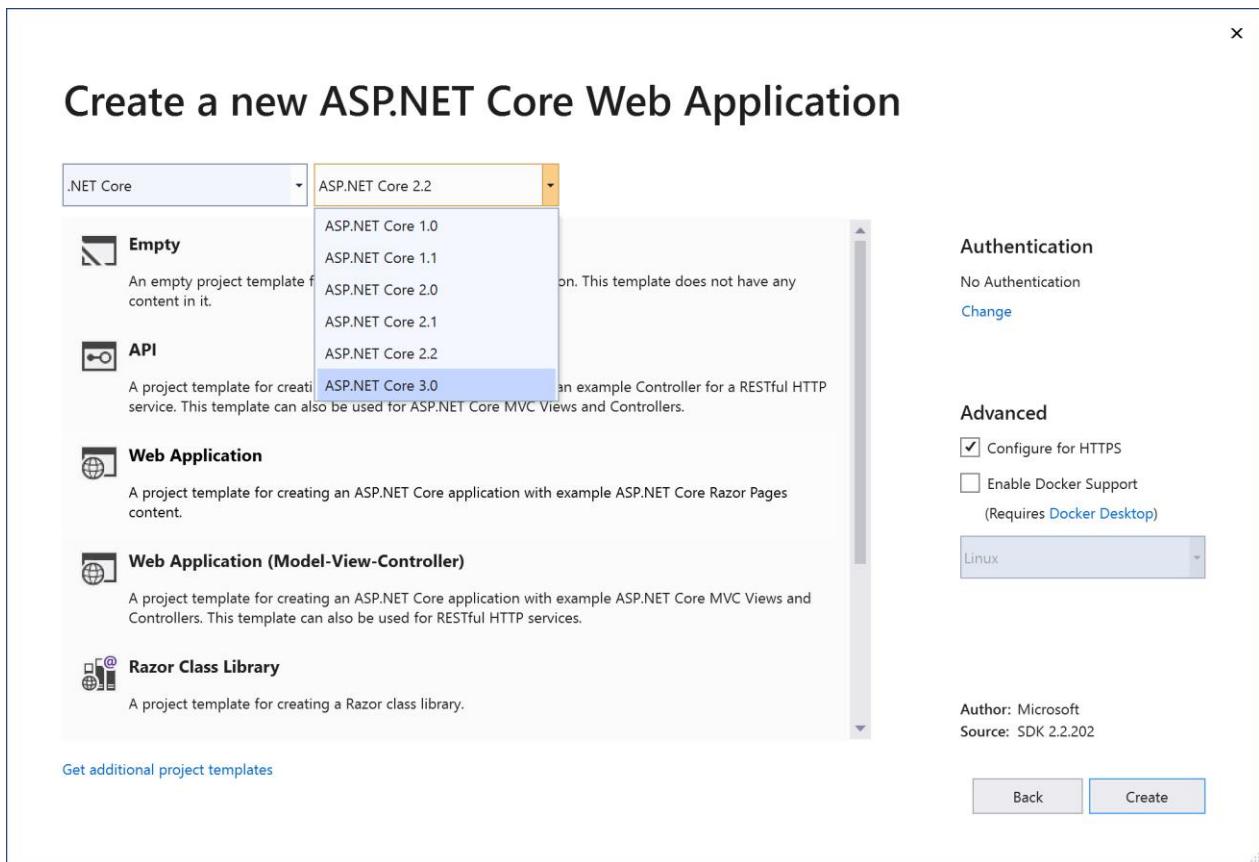
*But wait a minute, where is the option for ASP .NET Core 3.0? Why isn't it available for selection? The answer is simple: ASP .NET Core 3.0 is still in preview as of April 2019, after the release of Visual Studio 2019. In order to create ASP .NET Core 3.0 projects with VS2019, you should do the following (for now):*

- Download .NET Core 3.0: <https://dotnet.microsoft.com/download/dotnet-core/3.0>
- Enable preview releases of .NET Core
- Click **Tools | Options** in the top menu
- Expand **Projects and Solutions | .NET Core**

- Ensure that “Use previews of the .NET Core SDK” checkbox is checked



Start to create a new project again and you should now see an option for Core 3.0!



Following some Twitter feedback, here are my results from running MSBuild from a command line:

- Tweet: <https://twitter.com/shahedc/status/1115360210779017216>

Just tried this:

```
> MSBuild.exe c:opathprojfile.csproj -t:rebuild
```

Results:

Build succeeded.

0 Warning(s)

0 Error(s)

FYI, I used MSBuild.exe from C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\MSBuild\Current\Bin

— Shahed (on Leave) (@shahedC) April 8, 2019

Here is the specific command, setting the target to rebuild the project:

```
> MSBuild.exe c:\path\projfile.csproj -t:rebuild
```

On my development machine, I used MSBuild.exe from the following path:

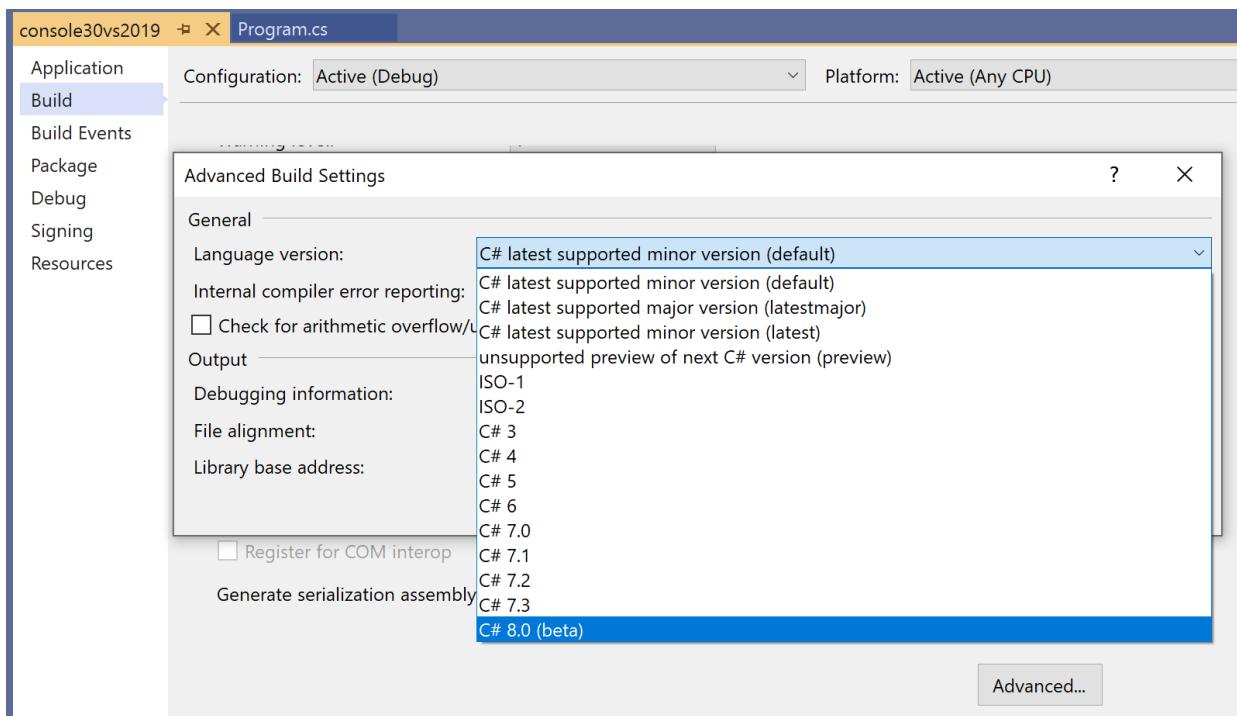
```
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\MSBuild\Current\Bin
```

## C# 8.0 Features

To ensure that C# 8.0 preview features are available, you can set the **LangVersion** property explicitly for your project. This setting is buried deep inside your Advanced settings within your project's Build tab.

To update the language setting:

1. Right-click your project in **Solution Explorer**.
2. Select **Properties** to view your project properties.
3. Click the **Build** tab within your project properties.
4. Click the **Advanced** button on the lower right.
5. Select the appropriate **Language version**, e.g. C# 8.0 (beta)
6. **Optional:** you may select “**unsupported preview...**” instead



The above screenshots show the aforementioned setting in the Visual Studio UI. If you wish to update your .csproj file directly, you may view/edit the <LangVersion> value. A few samples are shown below:

For a .NET Core 3.0 console app set to use C# *preview* versions, the value of <LangVersion> is set to the value “preview”:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
</Project>
```

For a .NET Core 3.0 console app set to use C# 8.0 explicitly, the value of <LangVersion> is set to the value “8.0”:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <LangVersion>8.0</LangVersion>
  </PropertyGroup>
```

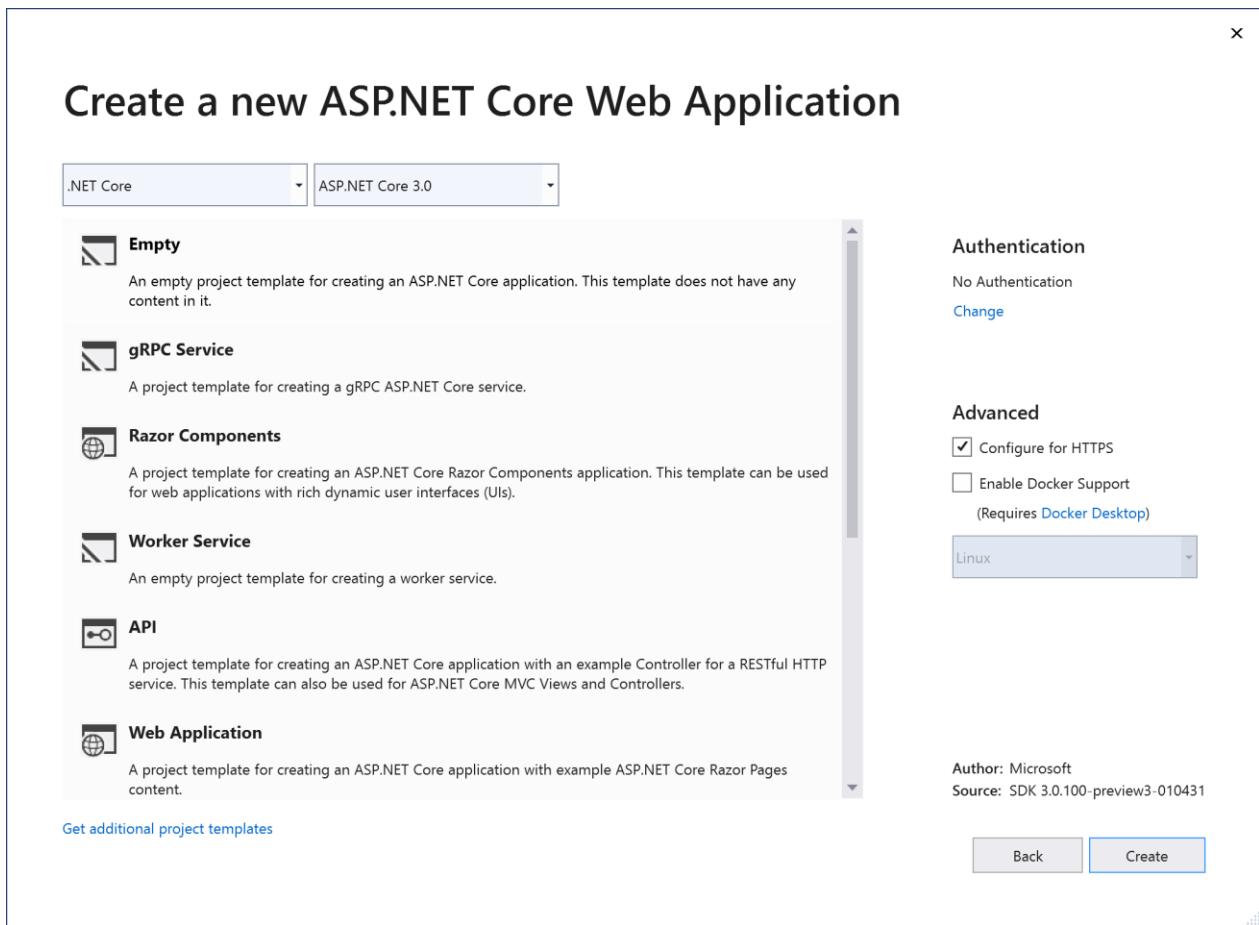
```
</Project>
```

Here is a list of C# 8.0 features, from recent preview releases:

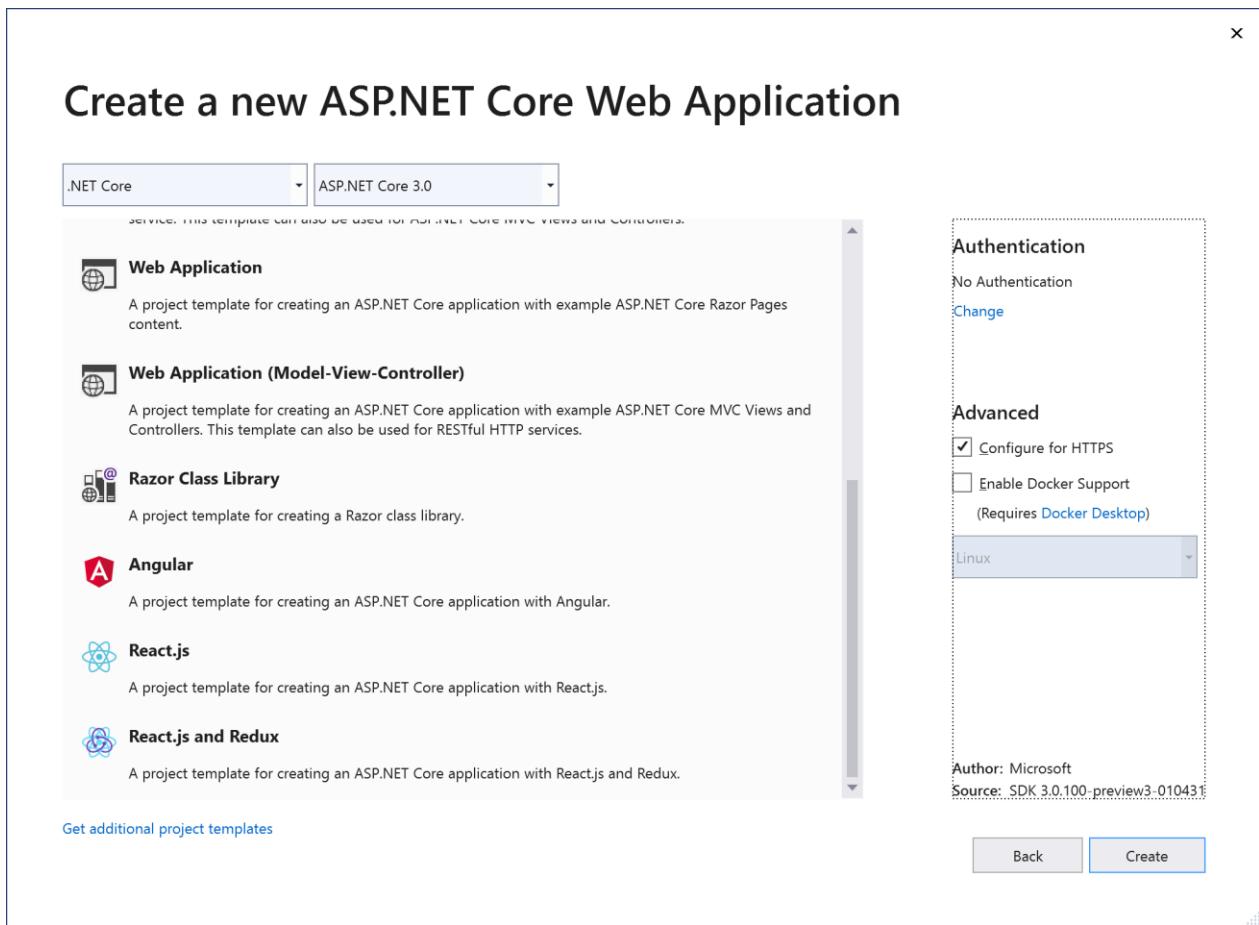
- Nullable reference types: design with intent, decide that some variables must always have a value, while others may be missing a value, using nullable reference types.
- Asynchronous streams: create and consume async streams, e.g. large streams of data
- Indices and ranges: Specify subranges of data with `Span<T>`, indicate indices in the subsets of data
- Pattern matching enhancements:
- Switch expressions: replace repetitive switch-case blocks with simpler *switch expressions*
- Property patterns: enhance switch statements by matching an object's properties
- Tuple patterns: use tuples for cases when matching sets of values
- Positional patterns: used to simplify the way we apply recursive patterns without having to name an object's properties
- Using declarations: used to simplify using blocks (within which an object is disposed when done) by disposing of the object at the end of the *enclosing* scope, i.e. its parent block.
- Static local functions: useful for local methods that are intended to be static.
- Disposable ref structs: allows the use of `Dispose()` methods to allow implementation of `IDisposable` in structs declared with a `ref` modifier.

## ASP .NET Core 3.0 Project Types

When you create a new Core 3.0 web project with Visual Studio 2019, you'll see some familiar project types. You will also see some *new* project types. These are shown in the 2 screenshots below:



Web Projects 1 of 2



Web Projects 2 of 2

The above project types are described below:

1. **Empty**: familiar empty project that just writes out Hello World to the HTTP Response, *without* the use of MVC or Razor Pages
2. **gRPC Service**: a *new* project type using Google's high-performance Remote Procedure Call (RPC) framework
3. **Razor Components**: initially called server-side Blazor, renamed to Razor Components to distinguish it from client-side Blazor, this will be once again be renamed to server-side Blazor again when ASP .NET Core 3.0 is released. Allows full-stack C# web app development.
4. **Worker Service**: a *new* project type that allows creation of background processes, e.g. Windows services or Linux daemons. May be relocated in the template list upon release.
5. **API**: familiar project type for creating Web APIs and RESTful services. Can be mixed and matched with Razor Pages or MVC components.

6. **Web Application**: familiar project type for creating Web Apps with Razor Pages. Can be mixed and matched with Web API and/or MVC components.
7. **Web Application (MVC)**: familiar project type for creating Web Apps with MVC application structure. Can be mixed and matched with Razor Pages and/or Web API.
8. **Razor Class Library**: relatively new project type for creating reusable UI Class Libraries with Razor Pages. See previous post on Razor Class Libraries.
9. **Angular, React.js, React.js and Redux**: familiar web projects for web developers who wish to build a JavaScript front-end, typically with a Web API backend.

Well, what about *client-side Blazor*? You may have noticed that server-side Blazor (aka Razor Components are mentioned, but there is no sign of client-side Blazor. As of April 2019, client-side Blazor running in the browser with WebAssembly is still experimental. As a result, it is not included with ASP .NET Core 3.0 but can be downloaded separately.

- Blazor Extension: <https://marketplace.visualstudio.com/items?itemName=aspnet.blazor>

For a quick refresher, check out my previous post on client-side Blazor:

- Blazor Full-Stack Web Dev in ASP .NET Core: <https://wakeupandcode.com/blazor-full-stack-web-dev-in-asp-net-core/>

If you need some help getting started, here's a handy guide from the official docs:

- Tutorial: Create an ASP.NET Core web app with Entity Framework and Visual Studio 2019: <https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-aspnet-core-ef-step-01?view=vs-2019>

## References

- What's new in .NET Core 3.0: <https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-core-3-0>

- A first look at changes coming in ASP.NET Core 3.0: <https://devblogs.microsoft.com/aspnet/a-first-look-at-changes-coming-in-asp-net-core-3-0/>
- ASP.NET Core updates in .NET Core 3.0 Preview 2: <https://devblogs.microsoft.com/aspnet/aspnet-core-3-preview-2/>
- ASP.NET Core updates in .NET Core 3.0 Preview 3: <https://devblogs.microsoft.com/aspnet/aspnet-core-updates-in-net-core-3-0-preview-3/>
- Blazor 0.9.0 experimental release now available: <https://devblogs.microsoft.com/aspnet/blazor-0-9-0-experimental-release-now-available/>
- Visual Studio 2019 Launch Event: <https://visualstudio.microsoft.com/vs2019-launch/>
- What's new in Visual Studio 2019: <https://docs.microsoft.com/en-us/visualstudio/ide/whats-new-visual-studio-2019?view=vs-2019>
- What's New in C# 8.0: <https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8>
- Little great things about Visual Studio 2019: <https://devblogs.microsoft.com/visualstudio/little-great-things-about-visual-studio-2019/>
- Do more with patterns in C# 8.0: <https://devblogs.microsoft.com/dotnet/do-more-with-patterns-in-c-8-0/>
- Get started with gRPC in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/grpc/grpc-start>

# Organizational Accounts for ASP .NET Core

By Shahed C on April 15, 2019

6 Replies

This is the **fifteenth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

- O is for Organizational Accounts
- Adding Authentication
- Configuring App Registration
- Using Authentication in your Code
- Endpoint Routing in MVC
- App Settings and Identity
- References

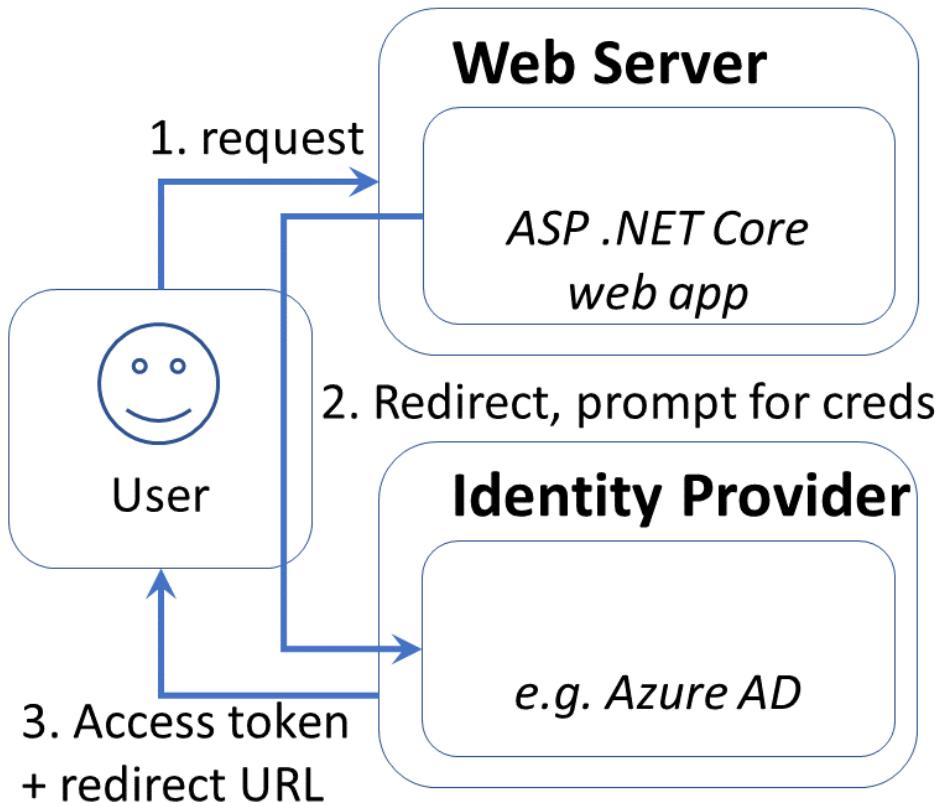
# O is for Organizational Accounts

If you've created new ASP .NET Core projects, you've probably seen an option to add authentication upon creation. In Visual Studio, the IDE provides radio buttons to select a specific type of Authentication. Using CLI commands (e.g. in the VS Code terminal) you can use the --auth flag to choose the type of authentication you'd like to add.

The possible values are:

- **None** – No authentication (Default).
- **Individual** – Individual authentication.
- **IndividualB2C** – Individual authentication with Azure AD B2C.
- **SingleOrg** – Organizational authentication for a single tenant.
- **MultiOrg** – Organizational authentication for multiple tenants.
- **Windows** – Windows authentication..

In this article, we will focus on the option for Work or School Accounts. This option can be used to authenticate users with AD (Active Directory, Azure AD or Office 365. In VS2019, a screenshot of the dialog is shown below:



To follow along, take a look at the sample project on Github:



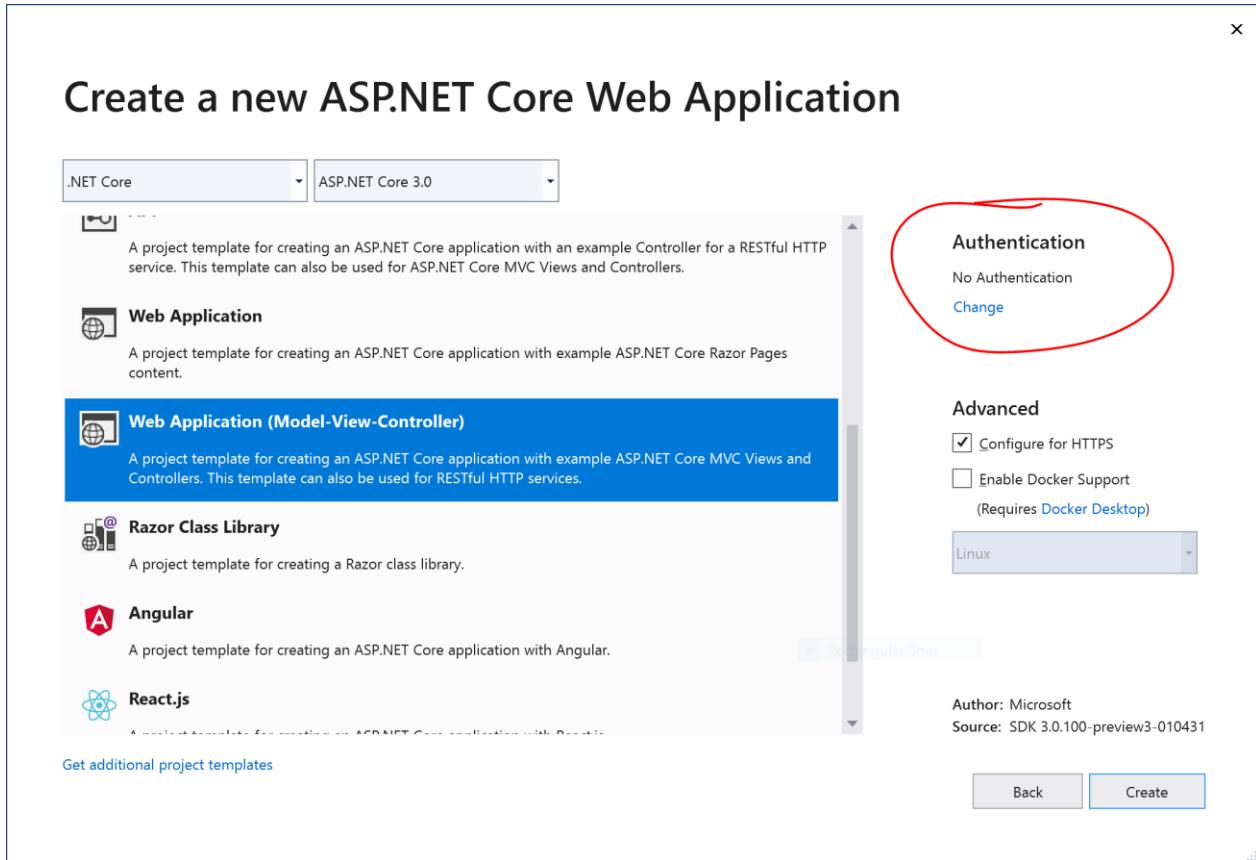
Org Authentication Sample: <https://github.com/shahedc/AspNetCore2019Org>

## Adding Authentication

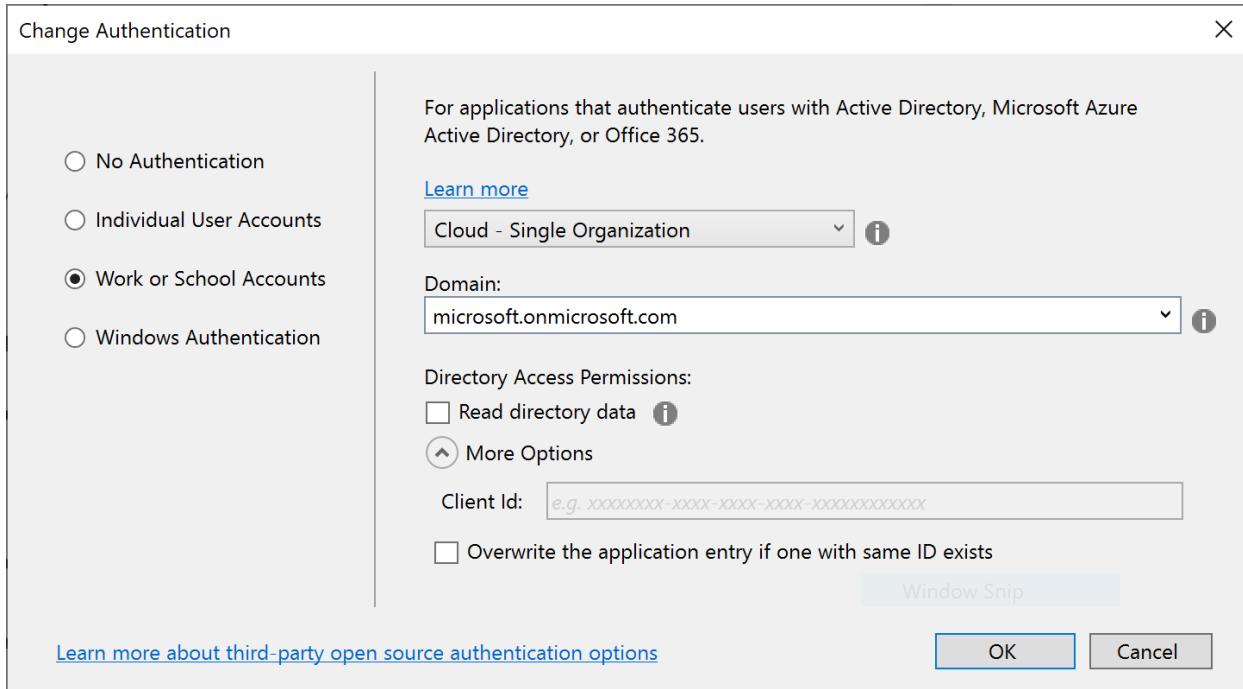
To add authentication to a new project quickly, here are the instructions for Visual Studio 2019.

If you choose to use the new splash screen:

1. Click “Create a new project”
2. Select “ASP .NET Core Web Application”
3. Click Next
4. Enter Project Name, Location, Solution Name
5. Optional: check the checkbox to place in the same directory
6. Click Create
7. Select .NET Core 3.0 and then ASP .NET Core 3.0 from dropdowns
8. Select a project type, e.g. Empty, Web Application (Razor Pages or MVC), etc
9. Click the “Change” action link in the Authentication section



This should allow you to change the authentication type to “Work or School Accounts” so that you may your organizational domain info. As always, you may select the little info buttons (lowercase i) to learn more about each field. Talk to your system administrator if you need more help on what domains to use.



**NOTE:** If you're having trouble locating .NET Core 3.0 project types in Visual Studio, take a look at the previous blog post in this series on .NET Core 3.0 to get some help on how to enable it.

If you wish to *skip* the Splash Screen instead upon launching VS2019:

1. Click “Continue without code” in the lower right area of the splash screen.
2. In the top menu, click File | New | Project (or Ctrl-Shift-N)
3. Follow the remaining steps outlined earlier in this section

To use CLI Commands in a Terminal window, use the **dotnet new** command followed by the --auth flag. The authentication type can be any of the aforementioned authentication types, e.g. Individual.

```
dotnet new mvc --auth Individual -o myproj
```

## Configuring App Registration

If you've used "Individual User Accounts" before, you've probably used a database to store user data, either on-premises or in the cloud. If you've used "Work or School Accounts" (i.e. organizational accounts), you may have the old App portal at the following URL:

- (OLD) Application Registration Portal: <https://apps.dev.microsoft.com>

The screenshot shows a web browser window for the Microsoft Application Registration Portal. The address bar displays the URL <https://apps.dev.microsoft.com/#/appList>. The page header includes the Microsoft logo and navigation links for 'Application Registration Portal', 'Tools', 'Docs', and 'Feedback'. A user profile icon is visible in the top right corner.

A prominent message banner states: "We will no longer support registering and managing converged and Azure AD applications here starting May 2019. We recommend that you manage your existing applications and register new applications by using the App registrations (Preview) experience in the Azure portal. [Click this banner to launch the preview experience.](#)"

## My applications

Converged applications [Learn More](#) [Add an app](#)

ⓘ We recommend registering and managing converged applications by using the App registrations (Preview) experience in the Azure Portal. [Go to the Azure portal](#)

Name	App ID / Client Id
https://apps.dev.microsoft.com/#	

You may see a message suggesting that you should go to the Azure Portal to use the *new* App Registrations feature. Although this feature is currently in preview, you can start using it right now. If you click the link, it should take you directly to the App Registrations page, and may prompt you to log in first.

The screenshot shows the Azure App Registration Overview page for an application named 'aspnetcore30'. The left sidebar contains navigation links for Overview, Quickstart, Manage, Branding, Authentication, Certificates & secrets, API permissions, Expose an API, Owners, Manifest, Support + Troubleshooting, Troubleshooting, and New support request. The main content area displays the app's details: Display name (aspnetcore30), Application (client) ID (redacted), Directory (tenant) ID (redacted), Object ID (redacted), and Supported account types (All Microsoft account users). It also includes sections for Endpoints, Call APIs (with icons for various Microsoft services like SharePoint, OneDrive, and Power BI), and Documentation (links to Microsoft identity platform, authentication scenarios, libraries, code samples, Microsoft Graph, glossary, and help and support). A banner at the bottom encourages users to 'View API Permissions'.

If you're not clear about how you got to this screen or how to come back to it later, here's a set of steps that may help.

1. Log in to the Azure Portal
2. Search for App Registrations
3. Arrive at the App Registrations page
4. If necessary, click the banner that takes you to the preview experience.

The screenshot shows the Azure App Registrations page under the 'App registrations' section. It features a search bar ('app registrations'), a 'Report a bug' button, and a 'Home > App registrations' breadcrumb. Below the header, there are three buttons: '+ New application registration', 'Endpoints', and 'Troubleshoot'. A prominent purple banner at the bottom states: 'The preview experience for App registrations is available. Click this banner to launch the preview experience.' with a right-pointing arrow icon.

The preview experience includes both your old and new app registrations. Click "New registration" to add a new app for authentication.

The screenshot shows the 'App registrations (Preview)' page. At the top, there's a breadcrumb navigation: Home > App registrations > App registrations (Preview). Below the title, there's a 'PREVIEW' link. A horizontal navigation bar includes 'New registration' (highlighted with a dashed blue border), 'Endpoints', 'Troubleshooting', and 'Got feedback?'. A blue banner at the bottom of the bar says, 'You are currently using the preview experience for App registrations. Click this banner to switch to the existing GA experience.' An information icon and a link to a guide are also present. Below the banner, a warning icon indicates that external users must register as first-party applications. At the bottom, there are two tabs: 'All applications' and 'Owned applications', with 'Owned applications' being the active tab.

In the form that follows, fill out the values for:

- **Name** (which you can change later)
- **Account Type** (your org, any org, any org + personal MSA)
- **Redirect URI** (where users will return after authentication)

## Register an application

PREVIEW

**!** If you are building an application for external users that will be distributed by Microsoft, you must register as a first party application to meet all security, privacy, and compliance policies. [Read our decision guide](#)

### \* Name

The user-facing display name for this application (this can be changed later).

SomeNewAppThatNeedsTobeRegistered



### Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (Microsoft)  
 Accounts in any organizational directory  
 Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)

[Help me choose...](#)

### Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web



e.g. <https://myapp.com/auth>

Now you should have options to configure your app and also download a pre-written application to get started. In the QuickStart section for your newly registered application (after selecting the guide for "ASP .NET Core"), you should see a button to make the changes for you and also download a configured Visual Studio application.

The screenshot shows the Microsoft Azure portal with the URL [https://ms.portal.azure.com/#blade/Microsoft\\_AAD\\_RegisteredApps/ApplicationMenuBlade/Quickstart/appId/fa758a39-1b2b-48d3-a592-c89123a46228/objectId/afa512cf-2561-4de...](https://ms.portal.azure.com/#blade/Microsoft_AAD_RegisteredApps/ApplicationMenuBlade/Quickstart/appId/fa758a39-1b2b-48d3-a592-c89123a46228/objectId/afa512cf-2561-4de...). The page title is "aspnetcore30 Registration" and the tab title is "Quickstart - Microsoft Azure". The main content area displays the "aspnetcore30 - Quickstart" blade, which is a preview for adding sign-in with Microsoft to an ASP.NET Core web app. It includes a sidebar with various service icons and a main content area with a diagram of the OAuth 2.0 flow, step-by-step instructions, and download links.

In the steps that follow:

1. Click the “Make the changes for me” button to make the necessary configuration changes.
2. Click the “Download” link to download the pre-configured Visual Studio solution.

#### Step 1: Configure your application in the Azure portal

For the code sample for this quickstart to work, you need to add reply URLs as <https://localhost:44321/> and <https://localhost:44321/signin-oidc>, add the Logout URL as <https://localhost:44321/signout-oidc>, and request ID tokens to be issued by the authorization endpoint.

[Make this change for me](#)

#### Step 2: Download your ASP.NET Core project

- [Download the Visual Studio 2017 solution](#)

#### Step 3: Configure your Visual Studio project

At the time of this writing, the project type is a VS2017 application. You can download it to inspect it, but I would recommend creating a new project manually in VS2019. There are some subtle differences

between projects created by VS2019 with authentication turned on, versus what you get with the downloaded project.

For further customization using the Manifest file available to you, check out the official documentation on the Azure AD app manifest:

- Understanding the Azure Active Directory app manifest: <https://docs.microsoft.com/en-us/azure/active-directory/develop/reference-app-manifest>

## Using Authentication in Your Code

When creating a new project in VS2019, you get the following lines of code in your **ConfigureServices()** method, including calls to **.AddAuthentication()** and **.addMvc()**.

```
// contents of ConfigureServices() when created in VS2019

services.AddAuthentication(AzureADDefaults.AuthenticationScheme)
    .AddAzureAD(options => Configuration.Bind("AzureAd", options));

services.AddMvc(options =>
{
    var policy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
    options.Filters.Add(new AuthorizeFilter(policy));
})
    .AddNewtonsoftJson();
```

If you download the pre-configured project from the Azure portal, you may notice an additional block of code in between **.AddAuthentication()** and **.addMVC()**.

```
// contents of ConfigureServices() when downloaded from Portal

services.AddAuthentication(AzureADDefaults.AuthenticationScheme)
    .AddAzureAD(options => Configuration.Bind("AzureAd", options));

services.Configure<OpenIdConnectOptions>(AzureADDefaults.OpenIdScheme,
    options =>
```

```

{
    options.Authority = options.Authority + "/v2.0/";
    options.TokenValidationParameters.ValidateIssuer = false;
} );

services.AddMvc(options =>
{
    var policy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
    options.Filters.Add(new AuthorizeFilter(policy));
})
.SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

```

This additional block calls `services.Configure<OpenIdConnectOptions>()` method to set up authentication for an Azure AD v2.0 endpoint. According to the documentation displayed in the Portal itself: *“All users with a work or school, or personal Microsoft account can use your application or API. This includes Office 365 subscribers. To change the supported accounts for an existing registration, use the manifest editor. Take care, as certain properties may cause errors for personal accounts.”*

You can copy the this block of code manually into your VS2019 project, and refer to the sample project's Startup.cs file.

```

services.Configure<OpenIdConnectOptions>(AzureADDefaults.OpenIdScheme,
options =>
{
    options.Authority = options.Authority + "/v2.0/";
    options.TokenValidationParameters.ValidateIssuer = false;
});

```

There is also a difference in the Compatibility Version setting in the code. The downloaded project for VS2017 currently sets compatibility for v2.1 but you can manually set this to 3.0 when you create a project manually in VS2019, as seen in this snippet from the sample Startup.cs file.

```

services.AddMvc(options =>
{
    var policy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
    options.Filters.Add(new AuthorizeFilter(policy));
})
.SetCompatibilityVersion(CompatibilityVersion.Version_3_0);

```

# Endpoint Routing in MVC

In the **Configure()** method of Startup.cs, the downloaded project contains familiar method calls to various Middleware components.

```
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseCookiePolicy();

app.UseAuthentication();

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

When you create an ASP .NET Core 3.0 project in VS2019, you may see the new *Endpoint Routing* feature, which makes it look like this:

```
app.UseHttpsRedirection();
app.UseStaticFiles();

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

app.UseCookiePolicy();

app.UseAuthentication();
app.UseAuthorization();
```

You may notice the addition of **app.UseRouting()** appearing in between **app.UseStaticFiles()** and **app.UseCookiePolicy**, and there is no explicit call to **app.UseMvc()**. This doesn't mean that the application is calling MVC ahead of time. Rather, the release notes explains it well:

*"So think of **UseRouting**(...) as making a deferred routing decision – where middleware that appear after it run in the middle. Any middleware that run after routing can see the results and read or modify the route data and chosen endpoint. When processing reaches the end of the pipeline, then the endpoint is invoked."*

You can read more about this in the writeup for v3.0 Preview 2:

- ASP.NET Core updates in .NET Core 3.0 Preview 2: <https://devblogs.microsoft.com/aspnet/aspnet-core-3-preview-2/>

## App Settings and Identity

In your appsettings.json file, you can set values for your App Settings. This includes the following **AzureAd** settings:

- Instance
- Domain
- ClientId
- TenantId
- CallbackPath

```
{  
  "AzureAd": {  
    "Instance": "https://login.microsoftonline.com/",  
    "Domain": "<YOUR_SUB_DOMAIN>.onmicrosoft.com",  
    "ClientId": "<YOUR_CLIENT_ID>",  
    "TenantId": "<YOUR_TENANT_ID>",  
    "CallbackPath": "/signin-oidc"  
  },  
  "Logging": {  
    "LogLevel": {  
      "Default": "Warning",  
      "Microsoft.Hosting.Lifetime": "Information"  
    }  
  },  
}
```

```
        "AllowedHosts": "*"
    }
```

You can see more information on appsettings.json values in the official docs. You can also open the appsettings.json from the portal-downloaded project to get your own app's Id values. The following documentation is specifically written for v2, but offers explanations for important fields such as **ClientId** and **TenantId**.

- Microsoft identity platform ASP.NET Core web app quickstart: <https://docs.microsoft.com/en-us/azure/active-directory/develop/quickstart-v2-aspnet-core-webapp>

To ensure that your authentication is working correctly, you can test it out by applying the **[Authorize]** attribute on a controller, e.g. the HomeController.cs class.

```
[Authorize]
public class HomeController : Controller
{
    //
}
```

To get a refresher on how to use the **[Authorize]** attribute, check out the post on Authorization from earlier in this blog series. For more on assigning users to specific roles, check out the official documentation at the following URL:

- Role-based authorization in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles>

Finally, take a look at the Login.partial.cshtml partial view to observe the way a user's identity is detected and shown. Here is a snippet from the sample:

```
@if (User.Identity.IsAuthenticated)
{
    <li class="nav-item">
        <span class="nav-text text-dark">Hello @User.Identity.Name!</span>
    </li>
}
```

Depending on what you have access to, the User.Identity object may not contain everything you expect it to. Here are some things to take note of:

- User.Identity should be null when *not* logged in

- User.Identity should be non-null when logged in...
- ... however, User.Identity.Name may be null even when logged in
- If User.Identity.Name is null, also check User.Identity.Claims
- User.Identity.Claims should have more than 0 values when logged in

The following screenshot shows an example of the user information in my debugging environment when logged in:



## References:

- ASP.NET Blog | ASP.NET Core updates in .NET Core 3.0 Preview 2: <https://devblogs.microsoft.com/aspnet/aspnet-core-3-preview-2/>
- Role-based authorization in ASP.NET Core | Microsoft Docs: <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles>
- Configure an application to access web APIs (Preview) | Microsoft Docs: <https://docs.microsoft.com/en-us/azure/active-directory/develop/quickstart-configure-app-access-web-apis>

- Understanding the Azure Active Directory app manifest | Microsoft Docs: <https://docs.microsoft.com/en-us/azure/active-directory/develop/reference-app-manifest>

# Production Tips for ASP .NET Core Web Apps

By Shahed C on April 22, 2019

2 Replies

This is the **sixteenth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**

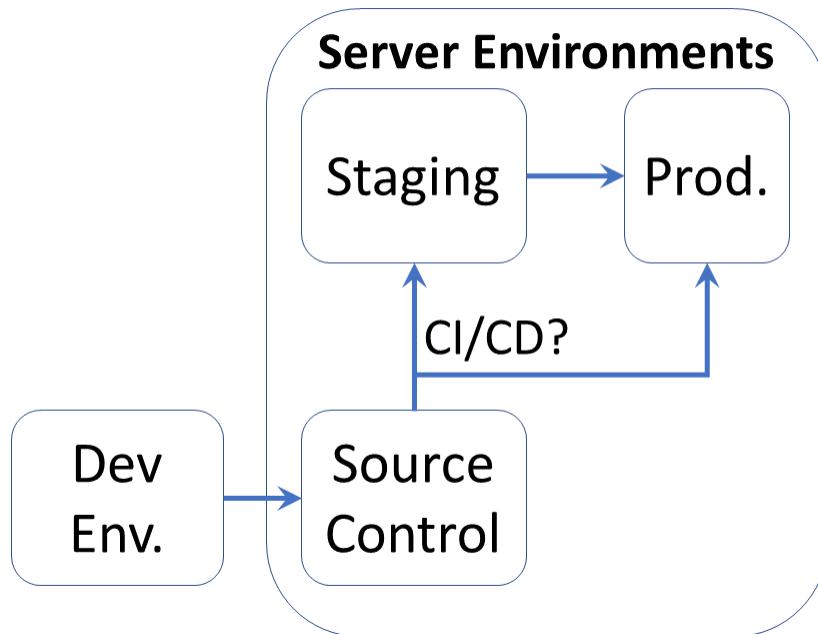


## In this Article:

- P is for Production Tips
- Deployment Slots
- Environment Configuration
- EF Core Migrations
- Scalability
- CI/CD
- Troubleshooting
- References

# P is for Production Tips

After getting through more than halfway in this A-Z series, this blog post takes a step back from application code to focus on production tips. Once you're ready to deploy (and maintain) your web app in production, there are many tips and tricks you should be aware of. In fact, feel free to discuss with your team members and the dev community to learn about other ways developers are deploying in production.



From development to server environments

While this article focuses on deployments to *Azure App Service*, you can use some of the lessons learned for your own environments. That being said, I would highly recommend taking a look at Azure for all your staging and production deployment needs.

## Deployment Slots

Azure makes it very easy to deploy your ASP .NET Core web application with the use of Deployment Slots. Instead of publishing a web app directly to production or worrying about downtime, you can publish to a Staging Slot and then perform a “swap” operation to essentially promote your Staging environment into Production.

**NOTE:** To enable multiple deployment slots in Azure, you must be using an App Service in a Standard, Premium, or Isolated tier.

If you need help creating a Web App in App Service, you may refer to my blog post on the topic:

- Deploying ASP .NET Core to Azure App Service: <https://wakeupandcode.com/deploying-asp-net-core-to-azure-app-service/>

To make use of deployment slots for your Web App:

1. Log in to the Azure Portal.
2. Create a new Web App if you haven't done so already.
3. Locate the App Service blade for your Web App
4. Enter the Deployment Slots item under Deployment
5. Click + Add Slot to add a new slot
6. Enter a Name, choose a source to clone settings (or not)
7. Click the Add button to create a new slot

The screenshot shows the 'Deployment slots' section of the Azure App Service portal. On the left, there's a navigation sidebar with links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Deployment (with Quickstart and Deployment slots selected), and Deployment Center. The main area has a 'Deployment Slots' heading with a sub-instruction: 'Deployment slots are live apps with their own hostnames. App content and configurations elements can be swapped between two deployment slots, including the production slot.' Below this is a table with the following data:

NAME	STATUS	APP SERVICE PLAN	TRAFFIC %
nonfreewebapp <span style="background-color: green; border: 1px solid black; padding: 2px;">PRODUCTION</span>	Running	NonFreeAsp	100
nonfreewebapp-nonfreestaging	Running	NonFreeAsp	0

You may now use the Swap feature to swap your deployed application between staging and production when the staged deployment is ready to be deployed into production. Note that all slots are immediately live at the specified endpoints, e.g. `hostname.azurewebsite.net`.

You may also adjust website traffic by setting the Traffic % manually. From the above screenshot, you can see that the Traffic % is initially set to 0 for the newly-created slot. This forces all customer traffic to go to the Production slot by default.

When deploying your application through various means (Visual Studio Publish, Azure CLI, CI/CD from your Source Control System, etc), you may choose the exact slot when there is more than one. You may also set up “Auto-Swap” to swap a slot (e.g. staging) automatically into production, upon pushing code into that slot.

To learn more about all of the above, check out the official docs at:

- Set up staging environments for web apps in Azure App Service: <https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots>

## Environment Configuration

To maintain unique configuration settings for each environment (e.g. staging database vs production database connection strings), you should have unique configuration settings for each environment. This is easily accomplished using the Configuration section in the Settings category of each slot's unique blade.

The screenshot shows the Azure portal interface for managing a web application. The left sidebar navigation includes Home, nonfreewebapp - Deployment slots, nonfreestaging (nonfreewebapp/nonfreestaging) - Configuration, Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Deployment (Quickstart, Deployment slots, Deployment Center), Settings (Configuration, Authentication / Authorizati..., Application Insights). The main content area is titled "nonfreestaging (nonfreewebapp/nonfreestaging) - Configuration". It has tabs for Application settings, General settings, Default documents, and Path mappings. The Application settings tab is active, showing a table for "Application settings" with one entry: WEBSITE\_NODE\_DEFAULT\_VERSION, Value: Hidden value. Click show values button above to view. Below that is a "Connection strings" section with a table showing StagingConnectionString, Value: Hidden value. Click show val, Type: SQLAzure. There are buttons for New application setting, Show values, and Advanced edit.

**NOTE:** If you need help with User Secrets for your development environment or Key Vault secrets for your server environment, consider the following posts from my 2018 series and earlier in this 2019 series:

- Your Web App Secrets in ASP .NET Core: <https://wakeupandcode.com/your-web-app-secrets-in-asp-net-core/>
- Key Vault for ASP .NET Core Web Apps: <https://wakeupandcode.com/key-vault-for-asp-net-core-web-apps/>

## EF Core Migrations

You may be wondering how you can deploy structural changes from your database into production. Perhaps, you write manual SQL scripts to run in production, maybe you use a tool to generate such SQL scripts or a combination of both. Many developers aren't aware but you can actually make use of Entity Framework Core (EF Core) Migrations to update your database structure.

To get a quick refresher on **EF Core Migrations** and **Relationships**, check out the following post:

- EF Core Migrations in ASP .NET Core: <https://wakeupandcode.com/ef-core-migrations-in-asp-net-core/>
- EF Core Relationships in ASP .NET Core: <https://wakeupandcode.com/ef-core-relationships-in-asp-net-core/>

You wouldn't typically run your "Update Database" command in production. Instead, you could generate a SQL Script from your EF Core Migrations. This will allow you to inspect the SQL Scripts (revise them if necessary), hand them over to a DBA if appropriate and finally run the SQL Scripts in production when required.

The following PowerShell command can be run in Visual Studio's Package Manager Console panel:

```
Script-Migration
```

The following CLI Command can be run on a Command Prompt, PowerShell prompt or VS Code Terminal window:

```
dotnet ef migrations script
```

You may set specific migrations to start from and/or end on:

```
Script-Migration -To <starting-migration>
Script-Migration -From <ending-migration>
```

You may also dump out the SQL scripts into a file for further inspection:

```
Script-Migration -Output "myMigrations.sql"
```

## Scalability

If you're deploying your web apps to Azure App Service, it's a no-brainer to take advantage of scalability features. You could ask Azure to scale your app in various ways:

- **Scale Up:** Upgrade to a more powerful (and higher priced) tier to add more CPU, memory and disk space. As you've seen with the appearance of staging slots, upgrading to a higher tier also provides additional features. Other features include custom domains (as opposed to just subdomains under `azurewebsites.net`) and custom certificates.
- **Scale Out:** Upgrade the number of VM instances that power your web app. Depending on your pricing tier, you can "scale out" your web app to dozens of instances.
- **Autoscaling:** When scaling out, you can choose when to scale out automatically:
  - Based on a Metric: CPU %, Memory %, Disk Queue Length, Http Queue Length, Data In and Data Out.
  - Up to a *specific* Instance Count: set a numeric value for the number of instances, set minimum and maximum.

An example of autoscaling on a metric could be: "*When the CPU% is >50%, increase instance count by 1*". When you had new scaling conditions, you may also set a schedule to start/end on specific dates and also repeated on specific days of the week.

The screenshot shows the Azure portal interface for managing an App Service plan. On the left, a sidebar lists 'Settings', 'Scale up (App Service plan)', 'Scale out (App Service plan)' (which is selected), and 'App Service plan'. The main area is titled 'Scale rule' and shows configuration for scaling out. It includes fields for 'Metric namespace' (App Service plans standard metrics), 'Metric name' (CPU Percentage), and a 'Time grain' of 1 minute. A table for 'DIMENSION NAME', 'OPERATOR', and 'DIMENSION VALUES' has a single row for 'Instance' with an '=' operator and 'All values' dimension value. Below this is a note about selecting multiple values for a dimension. A line chart displays 'CPU Percentage (Avg)' over time, with a specific point labeled '3.31 %'. Configuration options for 'Time grain (in mins)', 'Time grain statistic', 'Operator', and 'Threshold' are shown at the bottom. Buttons for 'Update' and 'Delete' are also present.

**NOTE:** In order to make use of Auto-Scaling, you'll have to upgrade to the appropriate tier to do so. You can still use Manual Scaling at a lower tier. Scalability features are not available on the F1 Free Tier.

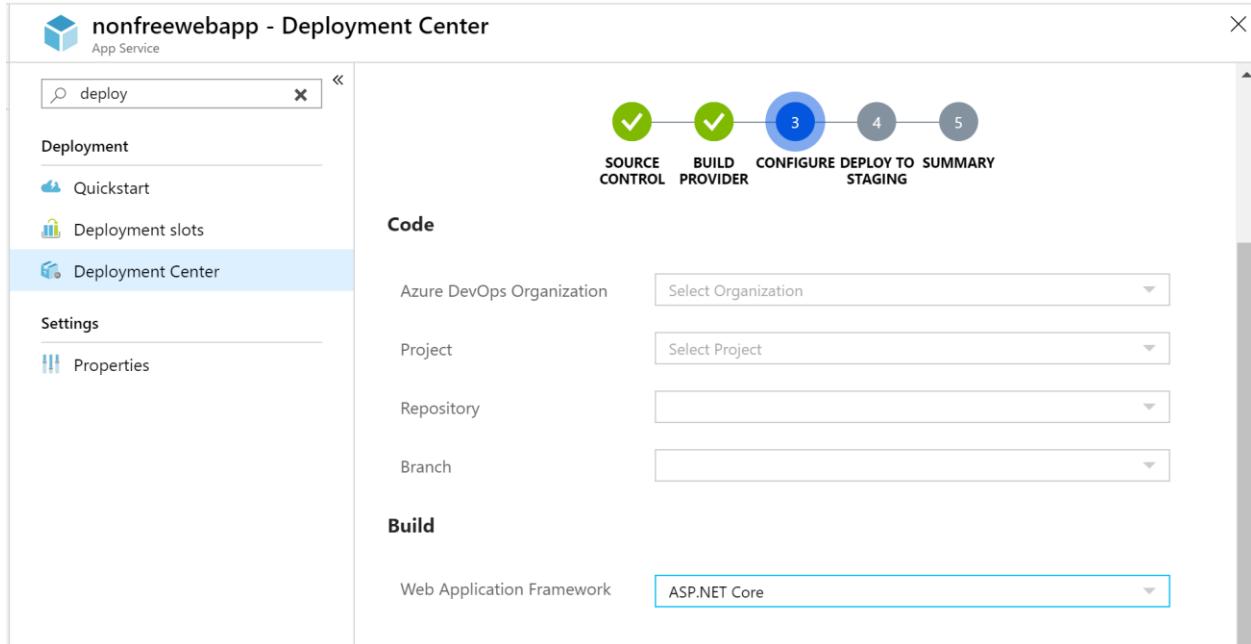
## CI/CD

There are countless possibilities to make use of CI/CD (*Continuous Integration and Continuous Deployment*) to make sure that your code has been merged properly, with unit tests passing and deployed into the appropriate server environments. Some of your options may include one of the following: Azure Pipelines, GitHub Actions, or some other 3rd party solution.

- Azure Pipelines: an offering of Azure DevOps services, you can quickly set up CI/CD for your web app, both public and private.

- GitHub Actions: available via GitHub, the new Actions feature allows you to automate your workflow

The **Deployment Center** feature in Azure's App Service makes it very easy to select Azure Pipelines (under Azure Repos) for your web app. This is all part of Azure DevOps Services, formerly known as VSTS (Visual Studio Team Services)



To get started with the above options, check out the official docs at:

- Azure Pipelines: <https://docs.microsoft.com/en-us/azure/devops/pipelines/?view=azure-devops>
- GitHub Actions: <https://github.com/features/actions>

TeamCity and Octopus Deploy are also popular products in various developer communities. Whatever you end up using, make sure you and your team select the option that works best for you, to ensure that you have your CI/CD pipeline set up as early as possible.

# Troubleshooting

Once your application has been deployed, you may need to troubleshoot issues that occur in Production. You can use a combination of techniques, including (but not limited to) Logging, Error Handling and Application Insights.

- **Logging:** From ASP .NET Core's built-in logging provider to customizable structured logging solutions (such as Serilog), logging helps you track down bugs in any environment.
- **Error Handling:** Anticipating errors before they occur, and then logging errors in production help you
- **Application Insights:** Enabled by default in Azure's App Service, Application Insights literally give you insight into your web application running in a cloud environment.

For more information on Logging and Error Handling, check out the earlier posts in this series:

- Logging in ASP .NET Core: <https://wakeupandcode.com/logging-in-asp-net-core/>
- Handling Errors in ASP .NET Core: <https://wakeupandcode.com/handling-errors-in-asp-net-core/>

For more information on *Application Insights*, stay tuned for an upcoming post in my next series that will focus on various Azure-related topics for ASP .NET Core developers.

## References

- Set up staging environments for web apps in Azure App Service: <https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots>
- Use multiple environments in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments>
- Configuration in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration>

- Migrations – EF Core: <https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/>
- Scale up features and capacities: <https://docs.microsoft.com/en-us/azure/app-service/web-sites-scale>
- Azure Pipelines Documentation: <https://docs.microsoft.com/en-us/azure/devops/pipelines/?view=azure-devops>

# Query Tags in EF Core for ASP .NET Core Web Apps

By Shahed C on April 29, 2019

1 Reply

This is the **seventeenth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**

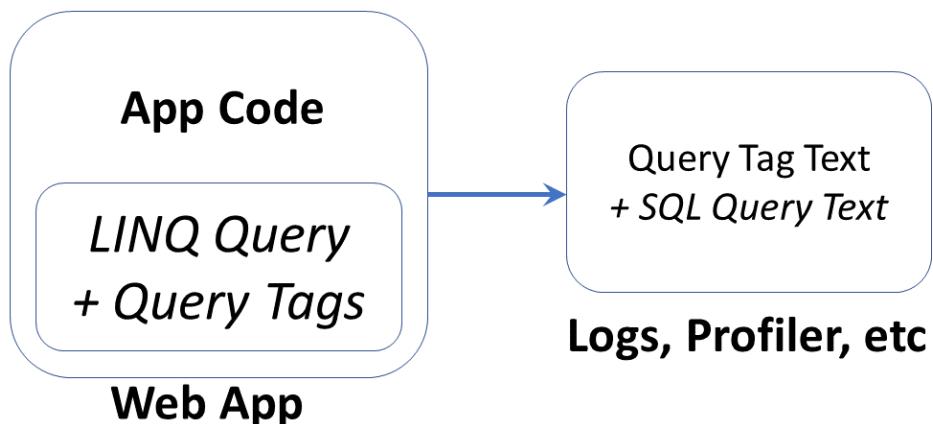


## In this Article:

- Q is for Query Tags in EF Core
- Implementing Query Tags
- Observing Query Tags in Logs
- Formatting Query Tag Strings
- References

## Q is for Query Tags in EF Core

Query Tags were introduced in Entity Framework (EF) Core 2.2, as a way to associate your LINQ Queries with SQL Queries. This can be useful when browsing log files during debugging and troubleshooting. This article explains how Query Tags work, how to find the output and how to format the text strings before displaying them.



**NOTE:** You may have read that Query Types have been renamed to entities without keys, but please note that Query *Types* (introduced in EF Core 2.1) are not the same thing as Query *Tags*.

As of ASP .NET Core 3.0 Preview 1, EF Core must be installed separately via NuGet (e.g. v3.0.0-preview4.19216.3), as it is no longer included with the ASP .NET Core shared framework. Also, the **dotnet ef** tool has to be installed as a global/local tool, as it is no longer part of the .NET Core SDK. For more information, see the official announcement for *Preview 4*, where it was first mentioned:

- Announcing Entity Framework Core 3.0 Preview 4:  
<https://devblogs.microsoft.com/dotnet/announcing-entity-framework-core-3-0-preview-4/>

## Implementing Query Tags

To follow along, take a look at the sample project on Github:



Query Tag Sample: <https://github.com/shahedc/WebAppWithQueries>

The sample includes a simple model called **MyItem**, with a few basic fields:

```
public class MyItem
{
    public int Id { get; set; }
    public string MyItemName { get; set; }
    public string MyItemDescription { get; set; }
}
```

A collection of **MyItem** objects are defined as a **DbSet** in the **ApplicationDbContext**:

```
public DbSet<WebAppWithQueries.Models.MyItem> MyItems { get; set; }
```

The **QueriedData()** action method in the **MyItemController** defines a Query Tag with the **TagWith()** method, as shown below:

```
public async Task<IActionResult> QueriedData()
{
    var topX = 2;
    var myItems =
        (from m in _context.MyItems.TagWith($"This retrieves top {topX} Items!")
         orderby m.Id ascending
         select m).Take(topX);

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

In the above query, the **TagWith()** method takes a single string value that can be stored along with wherever the resultant SQL Queries are logged. This may include your persistent SQL Server database logs or Profiler logs that can be observed in real-time. It doesn't affect what gets displayed in your browser.

Index - WebAppWithQueries

https://localhost:44305/MyItem

WebAppWithQueries Home Privacy My Items Queried Data More Tags Register Login

# Index

Create New

MyItemName	MyItemDescription	
item 1	item 1 description	Edit   Details   Delete
item 2	item 2 description	Edit   Details   Delete
item 3	item 3 description	Edit   Details   Delete

© 2019 - WebAppWithQueries - [Privacy](#)

## Observing Query Tags in Logs

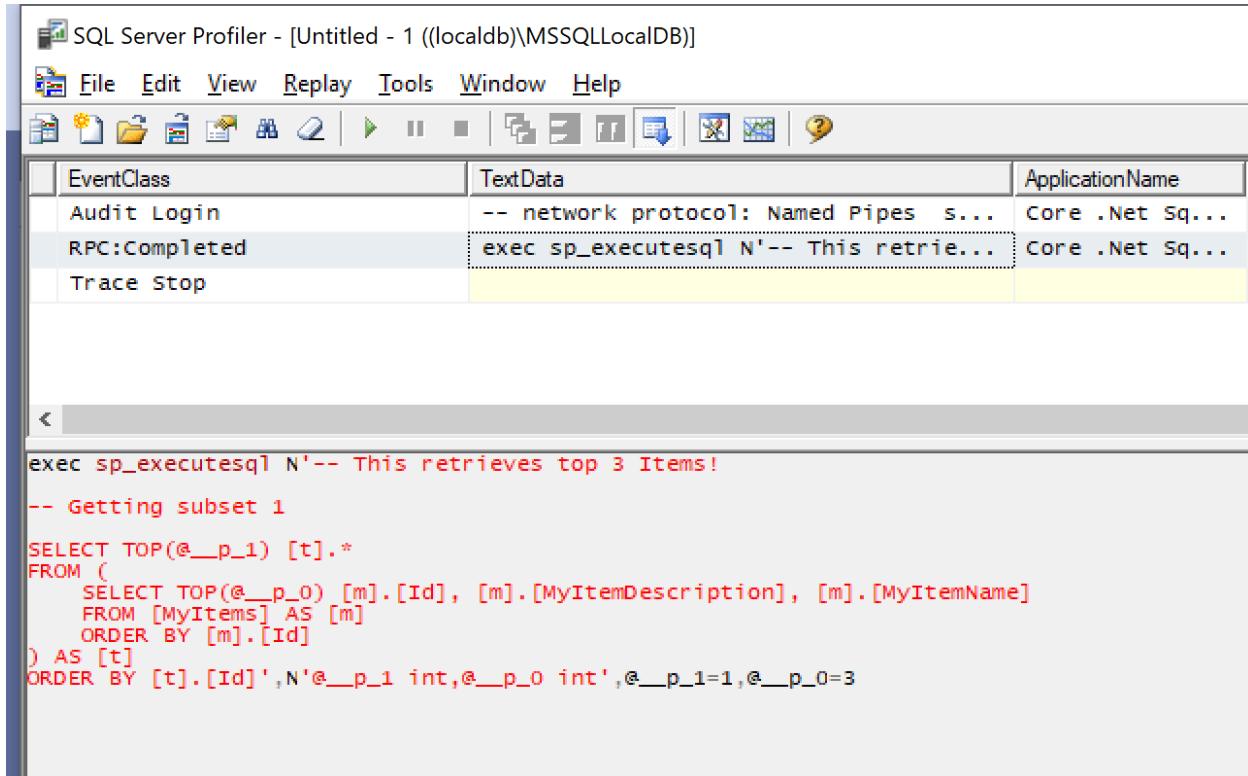
Using the SQL Server Profiler tool, the screenshot below shows how the Query Tag string defined in the code is outputted along with the SQL Query. Since topX is set to 2, the final string includes the value of topX inline within the logged text (more on formatting later).

The screenshot shows the SQL Server Profiler interface. The title bar reads "SQL Server Profiler - [Untitled - 1 ((localdb)\MSSQLLocalDB)]." The menu bar includes File, Edit, View, Replay, Tools, Window, and Help. The toolbar contains various icons for file operations and monitoring. A table view displays event data with columns: EventClass, TextData, and ApplicationName. The data shows several events from the Core .Net SqlClient Data Provider application, including Audit Logout, RPC:Completed (exec sp\_reset\_connection), Audit Login (network protocol: Named Pipes s...), another RPC:Completed (exec sp\_executesql N'-- This retrie...'), and Trace Stop. Below the table, a text area shows a SQL query:

```
exec sp_executesql N'-- This retrieves top 2 Items!
SELECT TOP(@__p_0) [m].[Id], [m].[MyItemDescription], [m].[MyItemName]
FROM [MyItems] AS [m]
ORDER BY [m].[Id]',N'@__p_0 int',@__p_0=2
```

From the code documentation, the **TagWith()** method “adds a tag to the collection of tags associated with an EF LINQ query. Tags are query annotations that can provide contextual tracing information at different points in the query pipeline.”

Wait a minute... does it say “collection of tags”...? Yes, you can add a collection of tags! You can call the method multiple times within the same query. In the **QueriedDataWithTags()** action of method the MyItemController class, you can call a string of methods to trigger cumulative calls to **TagWith()**, which results in multiple tags being stored in the logs.



## Formatting Query Tag Strings

You may have noticed that I used the \$ (dollar sign) symbol in my Query Tag samples to include variables inline within the string. In case you're not familiar with this language feature, the string interpolation feature was introduced in C# 6.

```
$"This retrieves top {topX} Items!"
```

You may also have noticed that the profiler is showing the first comment in the same line as the leading text “**exec sp\_executesql**” in the Profiler screenshot. If you want to add some better formatting (e.g. newline characters), you can use the so-called *verbatim identifier*, which is essentially the @ symbol ahead of the string.

```
@"This string has more
than 1 line!"
```

While this is commonly used in C# to allow newlines and unescaped characters (e.g. backslashes in file paths), some people may not be aware that you can use it in Query Tags for formatting. This operator allows you to add multiple newlines in the Query Tag's string value. You can combine both operators together as well.

```
@$"This string has more than 1 line  
and includes the {topX} variable!"
```

In an actual example, a newline produces the following results:

The screenshot shows the SQL Server Profiler interface. The top part displays a table with columns: EventClass, TextData, and ApplicationName. The table contains four rows corresponding to the events listed in the code above. The bottom part shows a large multi-line SQL script in a text editor window.

EventClass	TextData	ApplicationName
Audit Logout		Core .Net Sq...
RPC:Completed	exec sp_reset_connection	Core .Net Sq...
Audit Login	-- network protocol: Named Pipes s...	Core .Net Sq...
RPC:Completed	exec sp_executesql N'-- -- This r...'	Core .Net Sq...

```
exec sp_executesql N'--  
-- This retrieves top 3 Items!  
  
-- Getting subset 1  
  
SELECT TOP(@__p_1) [t].*  
FROM (  
    SELECT TOP(@__p_0) [m].[Id], [m].[MyItemDescription], [m].[MyItemName]  
    FROM [MyItems] AS [m]  
    ORDER BY [m].[Id]  
) AS [t]  
ORDER BY [t].[Id]',N'@__p_1 int,@__p_0 int',@__p_1=1,@__p_0=3
```

The above screenshot now shows the text from multiple Query Tags each on their own new line. As before, both of them were evaluated during the execution of a single SQL statement.

## References

- Query Tags – EF Core: <https://docs.microsoft.com/en-us/ef/core/querying/tags>
- Basic Queries – EF Core: <https://docs.microsoft.com/en-us/ef/core/querying/basic>
- Raw SQL Queries – EF Core: <https://docs.microsoft.com/en-us/ef/core/querying/raw-sql>

- Announcing Entity Framework Core 3.0 Preview  
4: <https://devblogs.microsoft.com/dotnet/announcing-entity-framework-core-3-0-preview-4/>

# Razor Pages in ASP .NET Core

By Shahed C on May 8, 2019

3 Replies

This is the **eighteenth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



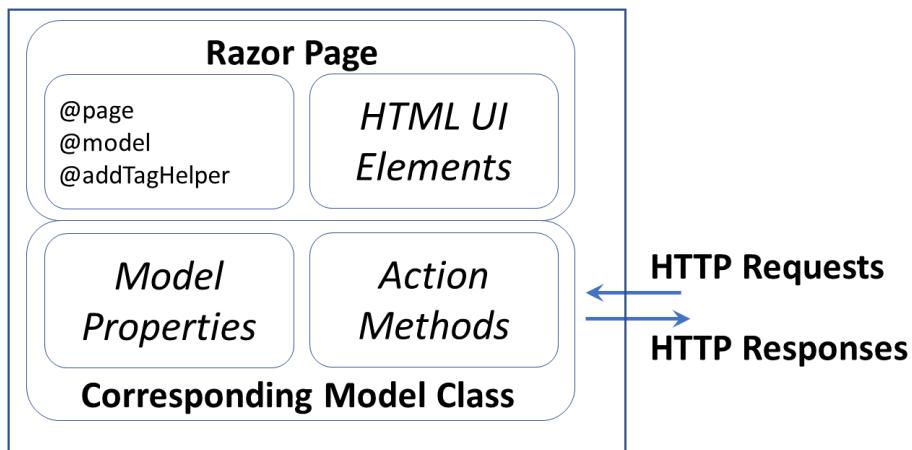
## In this Article:

- R is for Razor Pages
- Core 3.0 Packages
- Page Syntax
- Model Binding
- Page Parameters
- Page Routing
- Handler Methods
- References

# R is for Razor Pages

Razor Pages were introduced in ASP .NET Core v2.0, and briefly covered in my 2018 series. The post covered Pages in ASP .NET Core: Razor, Blazor and MVC Views. This post in the 2019 A-Z series will go deeper into Razor Pages and some of its features. You may also refer to a previous post to learn more about Forms and Fields (specifically the Razor Pages section).

Built on top of MVC in ASP .NET Core, Razor Pages allows you to simplify the way you organize and code your web apps. Your Razor Pages may coexist along with a backend Web API and/or traditional MVC views backed by controllers. Razor Pages are typically backed by a corresponding .cs class file, which represents a Model for the Page with Model Properties and Action Methods that represent HTTP Verbs. You can even use your Razor knowledge to work on Blazor fullstack web development.



## Core 3.0 Packages

To follow along, take a look at the sample project on Github:



Razor Pages (Core 3.0) Sample: <https://github.com/shahedc/RazorPagesCore30>

Let's start by taking a look at a 3.0 project (currently in preview) compared to a 2.x project (seen in 2018's post on Pages). The snippet below shows a .csproj for the sample app. This was created by starting with the Core 3.0 (Preview) Razor Pages Template in VS2019 and then following the official tutorial on Docs.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

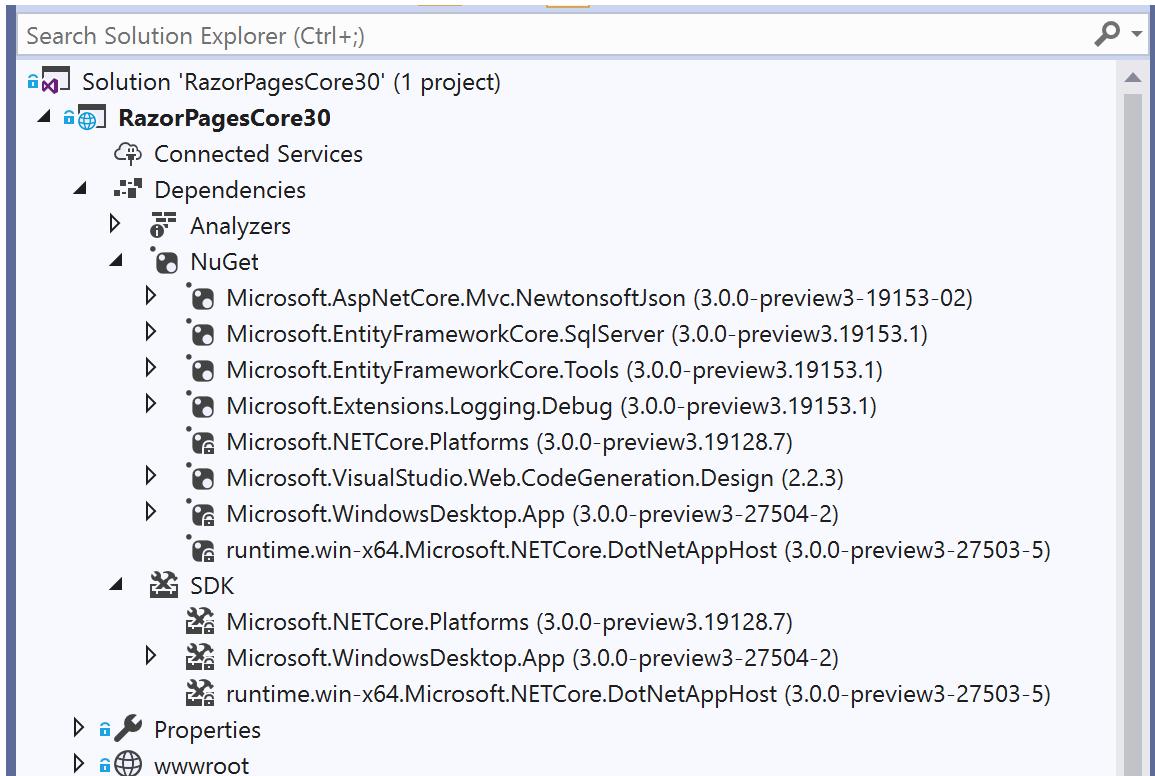
<PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
</PropertyGroup>

<ItemGroup>
    <PackageReference
        Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson" Version="3.0.0-preview3-19153-02" />
    <PackageReference
        Include="Microsoft.EntityFrameworkCore.SqlServer" Version="3.0.0-preview3.19153.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="3.0.0-preview3.19153.1">
        ...
    </PackageReference>
</ItemGroup>

</Project>
```

For ASP .NET Core 3.0, both **NewtonsoftJson** and **EF Core** have been removed from the ASP .NET Core shared framework. Instead, they are available as NuGet packages that can be included via **<PackageReference>** tags in the .csproj project file.

This is reflected in the Solution Explorer, where the Dependencies tree shows the **NewtonsoftJson** and **EF Core** packages nested under the NuGet node.



If you need a refresher on the new changes for ASP .NET Core 3.0, refer to the following:

- A first look at changes coming in ASP.NET Core 3.0: <https://devblogs.microsoft.com/aspnet/a-first-look-at-changes-coming-in-asp-net-core-3-0/>
- .NET Core 3.0, VS2019 and C# 8.0 for ASP .NET Core developers: <https://wakeupandcode.com/net-core-3-vs2019-and-csharp-8/#aspnetcore30>

## Page Syntax

To develop Razor Pages, you can reuse syntax from MVC Razor Views, including Tag Helpers, etc. For more information on Tag Helpers, stay tuned for an upcoming post in this series. The code snippet below shows a typical Razor page, e.g. Index.cshtml:

```
@page
@model IndexModel
{@
```

```

        ViewData["Title"] = "Home page";
    }

<!-- HTML content, with Tag Helpers, model attributes -->

```

Here is a quick recap of what a Razor Page is made of:

1. Each Razor Page starts with an **@page** directive to indicate that it's a *Razor Page*. This is different from *Razor Views* in MVC, which should not start with **@page**.
2. The **@page** directive may be followed by an **@model** directive. This identifies the corresponding C# model class, typically located in the same folder as the .cshtml page itself.
3. (Optional) You can include server-side code within an **@{}** block.
4. The rest of the page should include any HTML content you would like to display. This includes any server-side Tag Helpers and Model attributes.

Running the sample app shows the Movies Index page in action:

e.g. <https://localhost:44301/Movies>

Title	Release Date	Genre	Price	Rating	
Iron Man	5/2/2008	Action	\$19.99	PG-13	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Iron Man 2	5/7/2010	Action	\$19.99	PG-13	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Iron Man 3	5/3/2013	Action	\$19.99	PG-13	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Avengers: Endgame	4/26/2019	Epic Action	\$19.99	PG-13	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

© 2019 - RazorPagesCore30 - [Privacy](#)

# Model Binding

The .cs model class associated with the page includes both the model's attributes, as well as action methods for HTTP Verbs. In a way, it consolidates the functionality of an MVC Controller and C# viewmodel class, within a single class file.

The simplest way to use model binding in a Razor Page use to use the **[BindProperty]** attribute on properties defined in the model class. This may include both simple and complex objects. In the sample, the Movie property in the CreateModel class is decorated with this attribute as shown below:

```
[BindProperty]  
public Movie Movie { get; set; }
```

Note that **[BindProperty]** allows you to bind properties for **HTTP POST** requests by default. However, you will have to *explicitly* opt-in for **HTTP GET** requests. This can be accomplished by including an optional boolean parameter (**SupportsGet**) and setting it to True.

```
[BindProperty(SupportsGet = true)]  
public string SearchString { get; set; }
```

This may come in handy when passing in QueryString parameters to be consumed by your Razor Page. Parameters are optional and are part of the route used to access your Razor Pages.

To use the Model's properties, you can use the syntax **Model.Property** to refer to each property by name. Instead of using the name of the model, you have to use the actual word "Model" in your Razor Page code.

e.g. a page's model could have a complex object...

```
public Movie Movie { get; set; }
```

Within the complex object, e.g. the Movie class has a public ID property:

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

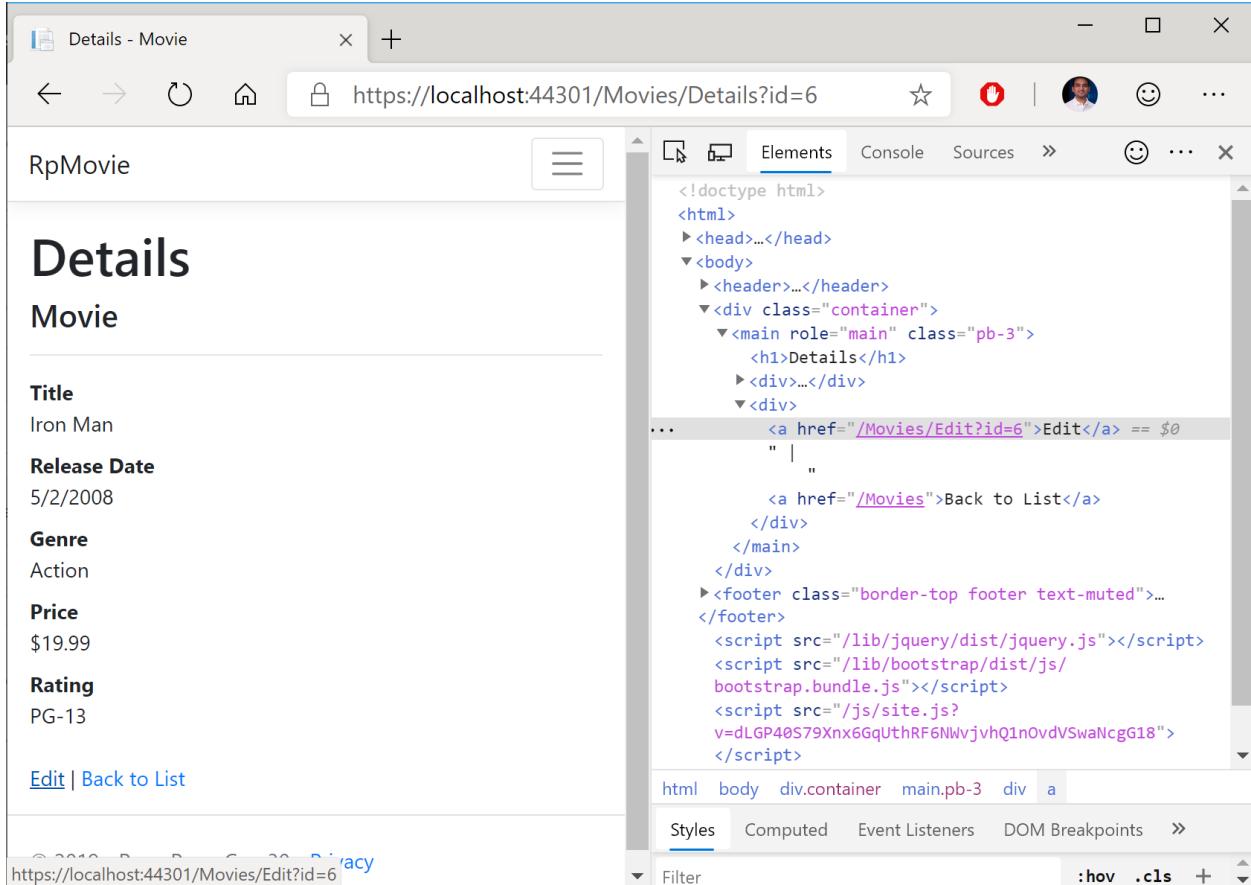
In the Razor Page that refers to the above model, you can refer to **Model.Movie.ID** by name:

```
@page  
@model RazorPagesCore30.Pages.Movies.DetailsModel
```

```
...  
<a asp-page=".~/Edit" asp-route-id="@Model.Movie.ID">Edit</a>
```

In this particular example, the `<a>` anchor tag is generated with a link to the Edit page with a route that uses a specific Movie ID value. The link points to the **Edit** page in the current subfolder (i.e. “Movies”), indicated by the period and slash in the path. The generated HTML looks like the following:

```
<a href="/Movies/Edit?id=6">Edit</a>
```



## Page Parameters

Page parameters can be included with the `@page` directive at the top of the page. To indicate that a parameter is optional, you may include a trailing ? question mark character after the parameter name. You may also couple the parameter names with a data type, e.g. `int` for integers.

```
@page "{id}"
```

```
@page "{id?}"
```

```
@page "{id:int?}"
```

The above snippet shows 3 different options for an id parameter, an optional id parameter and an integer-enforced id parameter. In the C# model code, a property named id can be automatically bound to the page parameter by using the aforementioned **[BindProperty]** attribute.

In the sample, the `SearchString` property in the `IndexModel` class for `Movies` shows this in action.

```
[BindProperty(SupportsGet = true)]  
public string SearchString { get; set; }
```

The corresponding page can then define an optional `searchString` parameter with the `@page` directive. In the HTML content that follows, Input Tag Helpers can be used to bind an HTML field (e.g. an input text field) to the field.

```
@page "{searchString?}"  
...  
Title: <input type="text" asp-for="SearchString" />
```

## Page Routing

As you may have guessed, a page parameter is setting up route data, allowing you to access the page using a route that includes the page name and parameter:

e.g. <https://servername/PageName/?ParameterName=ParameterValue>

In the sample project, browsing to the `Movies` page with the search string “Iron” shows a series of “Iron Man” movies, as shown in the following screenshot.

e.g. <https://localhost:44301/Movies/?SearchString=Iron>

Title	Release Date	Genre	Price	Rating	
Iron Man	5/2/2008	Action	\$19.99	PG-13	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Iron Man 2	5/7/2010	Action	\$19.99	PG-13	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Iron Man 3	5/3/2013	Action	\$19.99	PG-13	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

Here, the value for **SearchString** is used by the **OnGetAsync()** method in the `Index.cshtml.cs` class for the Movies page. In the code snippet below, you can see that a LINQ Query filters the movies by a subset of movies where the Title contains the **SearchString** value. Finally, the list of movies is assigned to the **Movie** list object.

```
...
public IList<Movie> Movie { get; set; }
...
public async Task OnGetAsync()
{
    ...
    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }
    ...
    Movie = await movies.ToListAsync();
}
```

By convention, all Razor Pages should be in a root-level “Pages” folder. Think of this “Pages” folder as a virtual root of your web application. To create a link to a Razor Page, you may link to the name of a Razor Page at the root level (e.g. “/Index”) or a Razor Page within a subfolder (e.g. “/Movies/Index”).

```

<a class="nav-link text-dark" asp-area="" asp-page="/Index">Home</a>

<a class="nav-link text-dark" asp-area="" asp-page="/Movies/Index">MCU Movies</a>

```



## Handler Methods

The `OnGetAsync()` method seen in the previous method is triggered when the Razor Page is triggered by an **HTTP GET** request that matches its route data. In addition to `OnGetAsync()`, you can find a complete list of *Handler Methods* that correspond to all HTTP verbs. The most common ones are for GET and POST:

- `OnGet()` or `OnGetAsync` for HTTP GET
- `OnPost()` or `OnPostAsync` for HTTP POST

When using the Async alternatives for each handler methods, you should return a `Task` object (or `void` for the non-async version). To include a return value, you should return a `Task<IActionResult>` (or `IActionResult` for the non-async version).

```

public void OnGet() {}
public IActionResult OnGet() {}
public async Task OnGetAsync() {}

public void OnPost() {}

```

```
public IActionResult OnPost() { }
public async Task<IActionResult> OnPostAsync() { }
```

To implement custom handler methods, you can handle more than one action in the same HTML form. To accomplish this, use the `asp-page-handler` attribute on an HTML `<button>` to handle different scenarios.

```
<form method="post">
<button asp-page-handler="Handler1">Button 1</button>
<button asp-page-handler="Handler2">Button 2</button>
</form>
```

To respond to this custom handlers, the exact handler names (e.g. Handler1 and Handler2) need to be included after `OnPost` in the handler methods. The snippet below shows the corresponding examples for handling the two buttons.

```
public async Task<IActionResult> OnPostHandler1Async()
{
    //...
}
public async Task<IActionResult> OnPostHandler2Info()
{
    // ...
}
```

**NOTE:** if you need to create a public method that you don't have to be recognized as a handler method, you should decorate such a method with the **[NonHandler]** attribute.

## References

- Introduction to Razor Pages in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/razor-pages/>
- Tutorial Overview: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/>
- Tutorial: Get started with Razor Pages in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start>
- Learn Razor Pages: <https://www.learnrazorpagedes.com/>
- Getting Started with Razor Pages: <https://visualstudiomagazine.com/articles/2019/02/01/getting-started-with-razor.aspx>

# Summarizing Build 2019 + SignalR Service for ASP .NET (Core) Developers

By Shahed C on May 14, 2019

1 Reply

This is the **nineteenth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

- S is for Summarizing Build 2019 (and SignalR Service!)
- Build 2019 for .NET Developers
- What's New in .NET Core 3.0 (Preview 5)
- What's New in ASP.NET Core 3.0 (Preview 5)
- What's Next for .NET Core (.NET Core vNext = .NET 5!)
- EF 6.3 for .NET Core, SqlClient & Diagnostics
- SignalR Service (Sneak Peek)
- References for Build 2019

- References for SignalR Service

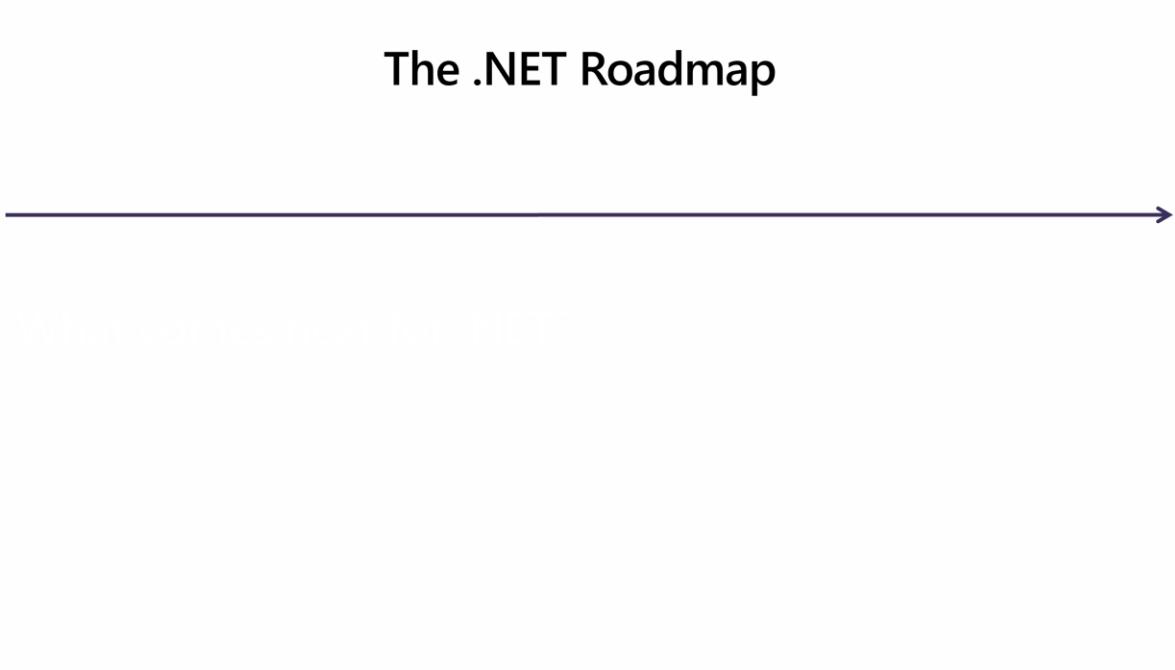
# S is for Summarizing Build 2019 (and SignalR Service!)

For the letter S, I was originally thinking of an article dedicated to SignalR Service. However, Microsoft's annual Build Conference just happened at the time of this writing. So this week's post will focus on Summarizing Build 2019 for ASP .NET (Core) Developers, followed by a sneak peek of SignalR Service at the end.

The biggest news for .NET Developers is that .NET Core is the future of .NET, going forward. Furthermore, .NET Core vNext will be named .NET 5, a unified platform for all .NET developers (.NET Framework, Xamarin/Mono and .NET Core).

The GIF below was generated from the .NET Overview session at Build 2019 to illustrate this future:

## The .NET Roadmap



GIF Source: <https://wakeupandcode.com/wp-content/uploads/2019/05/DotNet5-Next.gif>

.NET Roadmap, from 2014 through 2016, then 2019 and Beyond

## Build 2019 for .NET Developers

The quickest way to catch up on Build 2019 content is to watch all the relevant videos. But how do you know which ones to watch? Well, if you're a .NET developer, I've put together a handy video playlist specifically for .NET Developers (including ASP .NET Core Developers).

- Build 2019 for .NET Devs: <https://aka.ms/DotNet-Build2019>

Your **key takeaways** from new announcements should include:

- **.NET Core is the future of .NET:** If you've already started working with .NET Core, that's great! If you're starting a new project, you should consider .NET Core.
- **.NET Framework will continue to be supported:** If you have any existing applications on .NET Framework (Windows-only), you can keep those on .NET Framework.
- **.NET Releases will become more predictable:** Starting with .NET 5.0, there will be 1 major release every year, after which each even-numbered release (6.0, 8.0, etc) will come with LTS (Long-Term Support).

In 2019, the expected **schedule for .NET Core 3.x** is as follows:

- July 2019: .NET Core 3.0 RC (Release Candidate)
- September 2019: .NET Core 3.0 (includes ASP .NET Core 3.0)
- November 2019: .NET Core 3.1 (LTS)

In 2020 and beyond, the expected **schedule for .NET Core 5+** is shown below:

- Early to mid 2020: .NET 5.0 Preview 1
- November 2020: .NET 5.0
- November 2021: .NET 6.0 (LTS)
- November 2022: .NET 7.0
- November 2023: .NET 8.0 (LTS)

Minor releases (e.g. 5.1, etc) will be considered only if necessary. According to the official announcement, the first preview of .NET 5.0 should be available within the first half of 2020.

**NOTE:** The upcoming .NET 5.0 should not be confused with the so-called “ASP .NET 5” which was the pre-release name for ASP .NET Core 1.0 before the product was first released in 2016. Going forward, the name of the unified framework is simply .NET 5, without the need for a trailing “Core” in the name.

## What's New in .NET Core 3.0 (Preview 5)

As of May 2019, .NET Core 3.0 is in Preview 5, is expected to be in RC in July 2019, to be followed by a full release in September 2019. This includes ASP .NET Core 3.0 for web development (and more!). For my first look at .NET Core 3.0, you may browse through this earlier post in this series:

- .NET Core 3.0, VS2019 and C# 8.0 for ASP .NET Core developers: <https://wakeupandcode.com/net-core-3-vs2019-and-csharp-8/>

The primary themes of .NET Core 3.0 are:

1. **Windows desktop apps:** while this is usually not a concern for ASP .NET Core web application developers, it's good to know that Windows developers can start using .NET Core right away.
2. **Full-stack web dev:** Blazor is no longer experimental and its latest preview allows developers to use C# for full-stack web development. More info at: <https://blazor.net>

3. **AI & ML:** Not just buzzwords, Artificial Intelligence and Machine Learning are everywhere. ML.NET 1.0 is now available for C# developers to join this exciting new area of software development. More info at: [dot.net/ml](https://dot.net/ml)
4. **Big Data:** .NET for Apache Spark is now in Preview, available on Azure Databricks and Azure HDInsight. More info at: [dot.net/spark](https://dot.net/spark)

For more information on Blazor, you may browse through this earlier post in this series:

- Blazor Full-Stack Web Dev in ASP .NET Core: <https://wakeupandcode.com/blazor-full-stack-web-dev-in-asp-net-core/>

A lot has changed with Blazor in recent months, so the above post will be updated after Core 3.0 is released. In the meantime, check out the official Blazor session from Build 2019.

- **Blazor @ Build 2019:** <https://youtu.be/y7LAbdoNBJA>

## What's New in ASP.NET Core 3.0 (Preview 5)

What about ASP .NET Core 3.0 in Preview 5? In the Preview 5 announcement, you can see a handful of updates for this specific release. This includes the following:

- **Easy Upgrade:** to upgrade from an earlier preview, update the package reference in your .csproj project file to the latest version (3.0.0-preview5-19227-01)
- **JSON Serialization:** now includes support for reading/writing JSON using System.Text.Json. This is part of the built-in JSON support mentioned in the official writeup for .NET Core 3.0 Preview 5.
- **Integration with SignalR:** Starting with this release, SignalR clients and servers will now use the aforementioned System.Text.Json as the default Hub protocol. You can read more about this in the SignalR section of the Migration guide for 2.x to 3.0.

- **Continued NewtonsoftJson support:** In case you need to switch back to Newtonsoft.Json (previously the default option for the SignalR Hub), the instructions are provided in the aforementioned Migration guide and the announcement. Note that Newtonsoft.Json needs to be installed as a NuGet package.

There has been a lot of development in ASP .NET Core 3.0 in previous Preview releases, so you can refer to my ***earlier posts in the series*** for more info:

- [Dec 2018] Exploring .NET Core 3.0 and the Future of C#: <https://wakeupandcode.com/exploring-net-core-3-0-and-the-future-of-csharp/>
- [April 2019] .NET Core 3.0, VS2019 and C# 8.0 for ASP .NET Core developers: <https://wakeupandcode.com/net-core-3-vs2019-and-csharp-8/>

Here are ***links to all preview notes*** if you need a refresher on what was new in each Preview:

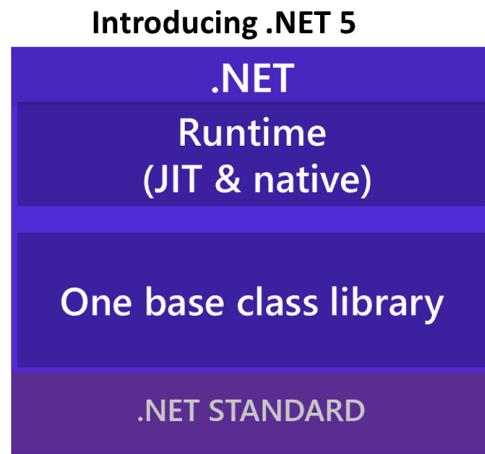
- [Jan 29, 2019] Preview 2: <https://devblogs.microsoft.com/aspnet/aspnet-core-3-preview-2/>
- [March 6, 2019] Preview 3: <https://devblogs.microsoft.com/aspnet/asp-net-core-updates-in-net-core-3-0-preview-3/>
- [April 18, 2019] Preview 4: <https://devblogs.microsoft.com/aspnet/asp-net-core-updates-in-net-core-3-0-preview-4/>
- [May 6, 2019] Preview 5: <https://devblogs.microsoft.com/aspnet/asp-net-core-updates-in-net-core-3-0-preview-5/>

**NOTE:** Changes from each preview to the next is usually cumulative. However, please note that **Blazor** went from experimental to preview in April 2019, and is *now called Blazor on both the client and server*. Previously, server-side Blazor was temporarily renamed to Razor Components, but then it was changed back to server-side Blazor.

## What's Next for .NET Core (.NET Core vNext = .NET 5!)

So, what's next for .NET Core? First of all, the next major version won't be called .NET Core, as we now know. With the upcoming release of .NET 5, you can now rest assured that all your investment into .NET Core will carry over into the future unified release.

Imagine taking the cross-platform .NET Core and bringing in the best of Mono for a single BCL (Base Class Library implementation). You may recall that .NET Standard was introduced when there were multiple versions of .NET (Framework, Mono/Xamarin and Core). Going forward, .NET Standard will continue to exist and .NET 5 will adhere to .NET Standard.



Compare the unified diagram with the various versions of .NET Framework, .NET Core and Mono over the years:

## .NET Version History

	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	— 2014	2015	2016	2017	— 2018	2019
<b>.Net Fmwk</b>	1.0	1.1		2.0	3.0	3.5			4.0		4.5, 4.5.1, 4.5.2	4.6, 4.6.1	4.6.2	4.7, 4.7.1 4.7.2	4.8		
<b>.NET Core</b>														1.0, 1.1	2.0, 2.1, 2.2	3.0	
<b>Mono</b>			1.0, 1.1		1.2		2.0	2.2, 2.4, 2.6	2.8	2.10	3.0, ... 3.10	3.12, 4.0		5.0, ... 5.18	5.20		

Note that not everything in the world of .NET today will make it into .NET 5. Not to worry though, as there are various recommended alternatives for all .NET developers. Take the following technologies, for example:

- **Web Forms:** As you may have noticed, ASP .NET Web Forms have not been ported to ASP .NET Core. Instead of Web Forms, developers may consider Blazor as their choice of web application development.
- **WCF:** Although Web API has been included in ASP .NET Core, there is no option for WCF. Going forward, you may use gRPC as an alternative.

Migration Guides for the above scenarios will be provided at a later date.

## EF 6.3 for .NET Core, SqlClient & Diagnostics

In addition to ASP .NET Core itself, there are other tools and technologies that may be useful for ASP .NET Core developers. That may include (but is not limited to) the following:

- **Entity Framework 6.3:** In addition to the EF Core running on .NET Core, EF 6.x was known to run on the Windows-only .NET Framework 4.x but not on .NET Core. Going forward, EF 6.3 will run on .NET Core 3.0 across platforms.
- **SqlClient:** Instead of replacing the existing System.Data.SqlClient package directly, the new Microsoft.Data.SqlClient (available on NuGet) will support both .NET Core and .NET Framework.
- **.NET Core Diagnostic Tools:** Making use of .NET Core Global Tools, a new suite of tools will help developers with diagnostics and troubleshooting of perf issues.

From the tools' GitHub page, the following tools are currently available, with the following descriptions:

- **dotnet-dump:** *"Dump collection and analysis utility."*
- **dotnet-trace:** *"Enable the collection of events for a running .NET Core Application to a local trace file."*
- **dotnet-counters:** *"Monitor performance counters of a .NET Core application in real time."*

## SignalR Service (Sneak Peek)

Finally, let's take a quick peek at the all-new SignalR Service.

- **What:** SignalR Service is a cloud-based service available in Azure, to help developers add real-time features in web applications and more.
- **When:** SignalR Service has been available since Build 2018
- **Where:** Available in the Azure Portal
- **How:** Create a new "SignalR Service" Resource
- Direct link: <https://ms.portal.azure.com/#create/Microsoft.SignalRGalleryPackage>

The screenshot shows the Microsoft Azure portal interface. The left sidebar has a dark theme with various service icons like Bot Services, Function Apps, Resource groups, etc. The main content area is titled 'SignalR' and shows a form for creating a new resource. The fields are as follows:

- \* Resource Name: mysignalr2019
- \* Subscription: (dropdown menu)
- \* Resource group: Select existing... or Create new
- \* Location: East US
- \* Pricing tier: Free
- \* Unit count: 1 (with a slider)
- ServiceMode: Default

At the bottom are two buttons: 'Create' (in blue) and 'Automation options'.

- **Who** can use this: Web developers who want to build real-time features can get started with a variety of official Quickstart guides: ASP .NET Core, JavaScript, C#, Java and REST API

If you're already familiar with using SignalR, switching to using Azure SignalR Service is as easy as 1-2-3.

1. Append a call to `.AddAzureSignalR()` to `AddSignalR()` in the `ConfigureServices()` method of your Startup class.

2. Replace the call to **UseSignalR()** with a call to **UseAzureSignalR()** in your **Configure()** method
3. Ensure that your environment's connection string is set correctly for the key "Azure:SignalR:ConnectionString".

In the **ConfigureServices()** method, this is what your code should look like:

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

In the **Configure()** method, this is what your code should look like:

```
app.UseAzureSignalR(routes =>
{
    routes.MapHub<HubClassName> ("/HubRouteName");
});
```

Your connection string (in your environment or User Secrets) may look like the following:

```
"Azure:SignalR:ConnectionString":  
"Endpoint=<yourendpoint>;AccessKey=<yourkey>;"
```

For a detailed tutorial for ASP .NET Core developers, try this official guide:

- Quickstart to learn how to use Azure SignalR Service: <https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-quickstart-dotnet-core>

After the A-Z weekly series is complete, stay tuned for monthly blog posts about cool things .NET developers can do in Azure. This will include a more in-depth look at SignalR Service in a future writeup, including guidance for both web and mobile developers.

## References for Build 2019 Announcements

- Build 2019: <https://mybuild.techcommunity.microsoft.com/sessions>

- YouTube Playlist: <https://aka.ms/DotNet-Build2019>
- Announcing .NET Core 3.0 Preview 5: <https://devblogs.microsoft.com/dotnet/announcing-net-core-3-0-preview-5/>
- .NET Core is the Future of .NET: <https://devblogs.microsoft.com/dotnet/net-core-is-the-future-of-net/>
- Introducing .NET 5: <https://devblogs.microsoft.com/dotnet/introducing-net-5/>
- ASP.NET Core updates in .NET Core 3.0 Preview 5: <https://devblogs.microsoft.com/aspnet/asp-net-core-updates-in-net-core-3-0-preview-5/>
- Announcing Entity Framework 6.3 Preview with .NET Core Support: <https://devblogs.microsoft.com/dotnet/announcing-entity-framework-6-3-preview-with-net-core-support/>
- Introducing diagnostics improvements in .NET Core 3.0: <https://devblogs.microsoft.com/dotnet/introducing-diagnostics-improvements-in-net-core-3-0/>
- dotnet/diagnostics: <https://github.com/dotnet/diagnostics>
- Introducing the new Microsoft.Data.SqlClient: <https://devblogs.microsoft.com/dotnet/introducing-the-new-microsoftdatasqlclient/>

## References for SignalR Service

- Azure SignalR Service now supports ASP.NET!: <https://devblogs.microsoft.com/aspnet/azure-signalr-service-now-supports-asp-net/>
- Quickstart to learn how to use Azure SignalR Service: <https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-quickstart-dotnet-core>
- Tutorial: Azure SignalR Service authentication with Azure Functions: <https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-tutorial-authenticate-azure-functions>
- Guide for authenticating Azure SignalR Service clients: <https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-concept-authenticate-oauth>
- Azure SignalR Service serverless quickstart: <https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-quickstart-azure-functions-csharp>



# Tag Helper Authoring in ASP .NET Core

By Shahed C on May 21, 2019

Leave a reply

This is the **twentieth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

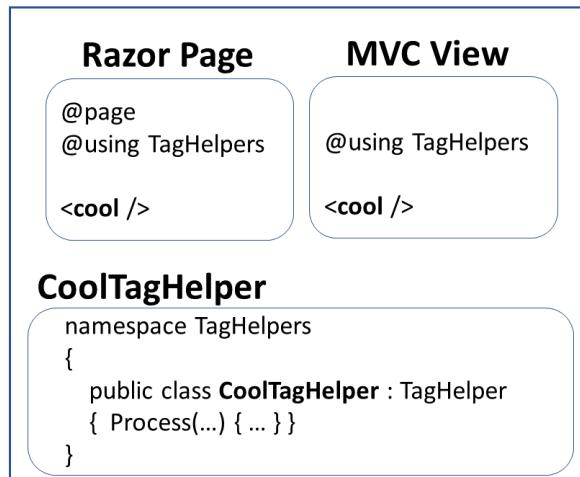
- T is for Tag Helper Authoring
- Custom Tag Helpers
- Setting Attributes and Content
- Updating Pre/Post Content
- Passing Complex Objects
- Handling Conditions
- References

# T is for Tag Helper Authoring

Tag Helpers are very useful for ASP .NET Core developers in creating HTML elements with server-side attributes. They work equally well in both Razor Pages and MVC views. Better yet, the syntax allows a front-end developer to easily customize the UI, with HTML/CSS knowledge.

If you need a refresher on built-in tag helpers in ASP .NET Core, you may revisit an earlier post in this series:

- Forms and Fields in ASP .NET Core: <https://wakeupandcode.com/forms-and-fields-in-asp-net-core/>



Authoring your own tag helpers is as easy as implementing the `ITagHelper` interface. To make things easier, the `TagHelper` class (which already implements the aforementioned interface) can be extended to build your custom tag helpers.

To follow along, take a look at the sample project on Github:



Tag Helper Authoring Sample: <https://github.com/shahedc/TagHelperAuthoring30>

**CAUTION:** the sample code contains spoilers for Avengers: Endgame (2019).

## Custom Tag Helpers

As with most concepts introduced in ASP .NET Core, it helps to use named folders and conventions to ease the development process. In the case of Tag Helpers, you should start with a “TagHelpers” folder at the root-level of your project for your convenience. You can save your custom tag helper classes in this folder.

This blog post and its corresponding code sample builds upon the official tutorial for authoring tag helpers. While the official tutorial covers instructions for MVC views, this blog post takes a look at a Razor Page example. The creation of Tag Helpers involves the same process in either case. Let’s start with the *synchronous* and *asynchronous* versions of a Tag Helper that formats email addresses.

The class EmailTagHelper.cs defines a tag helper that is a subclass of the TagHelper class, saved in the “TagHelpers” folder. It contains a **Process()** method that changes the output of the HTML tag it is generating.

```
public class EmailTagHelper : TagHelper
{
    ...
    // synchronous method, CANNOT call output.GetChildContentAsync();
    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        ...
    }
}
```

The class AsyncEmailTagHelper.cs defines a tag helper that is also a subclass of the TagHelper class, saved in the aforementioned “TagHelpers” folder. It contains a **ProcessAsync()** method, which has a different signature (returns Task object instead of void) and grabs the child content from the output using `output.GetChildContentAsync();`

```
public class AsyncEmailTagHelper : TagHelper
{
    ...
}
```

```

    // ASYNC method, REQUIRED to call output.GetChildContentAsync();
    public override async Task ProcessAsync(TagHelperContext context,
    TagHelperOutput output)
    {
        // ...
    }
}

```

In order to use the tag helper in a Razor Page, simply add a using statement for the Tag Helper's namespace, and then include a custom HTML tag that has the same name as the Tag Helper's class name (without the TagHelper suffix). For the Email and AsyncEmail Tag Helpers, the corresponding tags in your Razor Page would be <email> and <async-email> respectively.

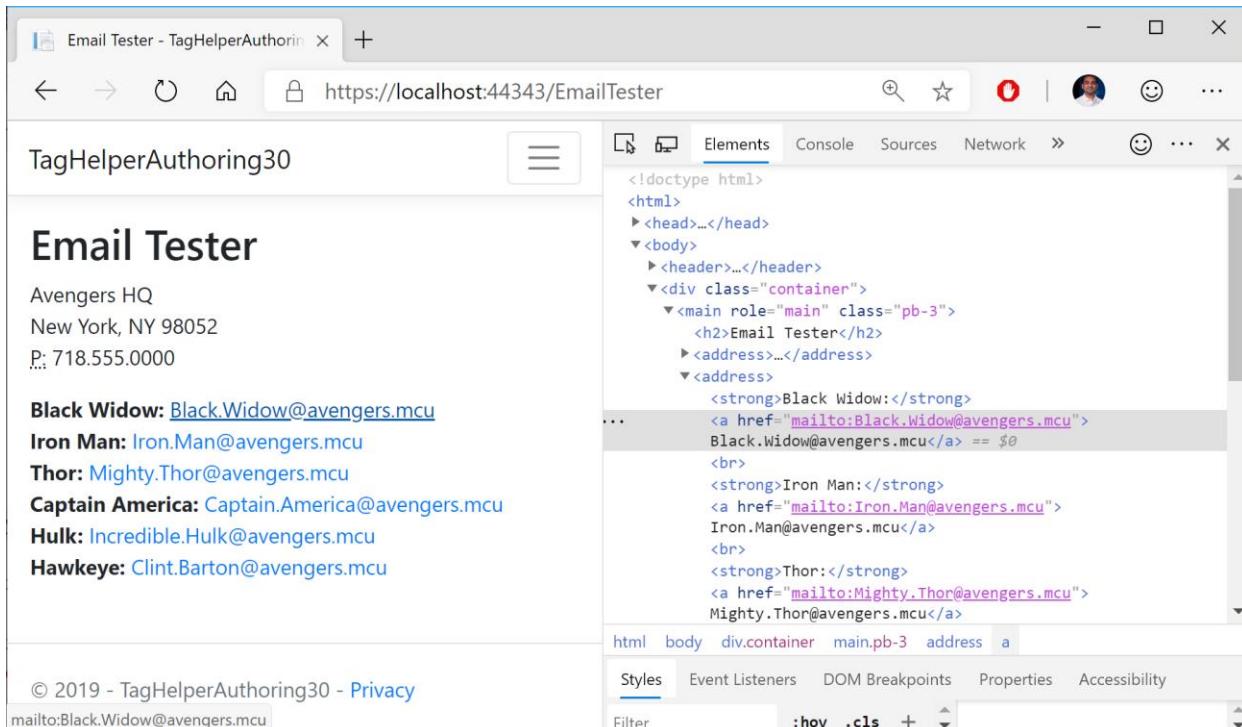
In the EmailTester.cshtml page:

```
<email mail-to="Black.Widow"></email>
```

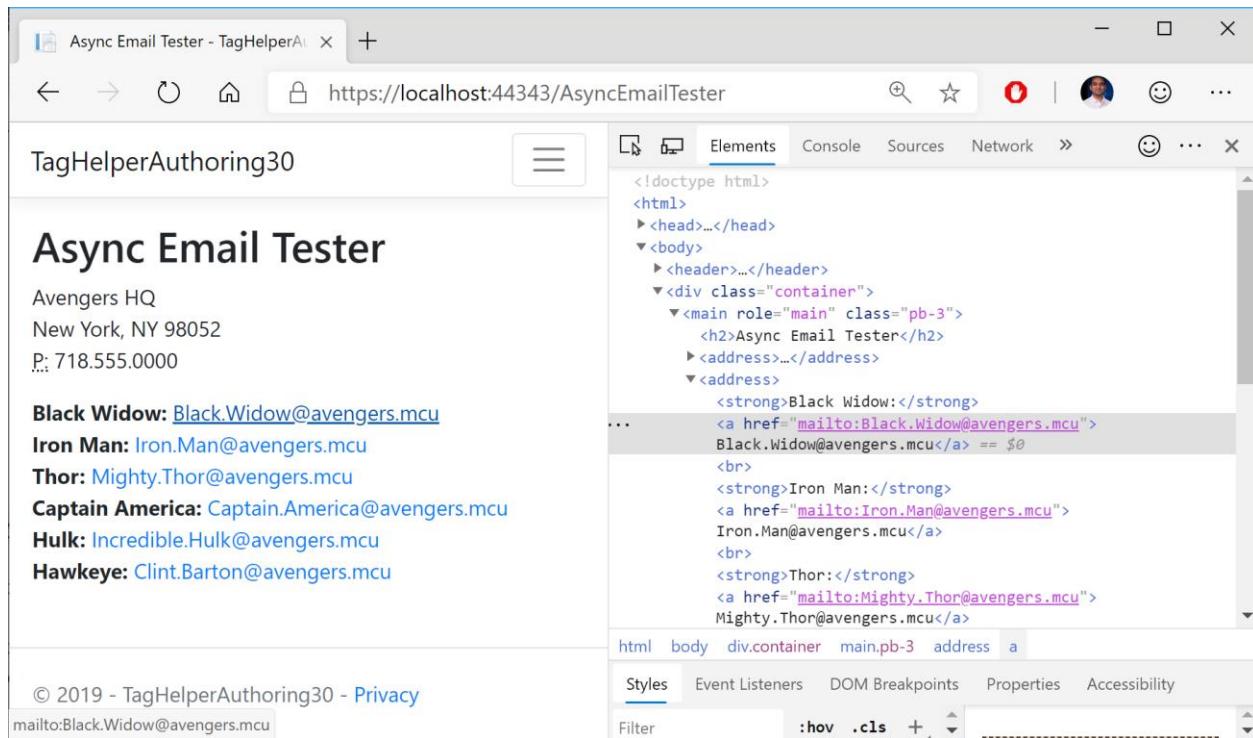
In the AsyncEmailTester.cshtml page:

```
<async-email>Black.Widow</async-email>
```

Note that the PascalCase capitalization in the class name corresponds to a lowercase tag in kebab-case. In a browser, the end result includes a clickable email link from the Razor Pages. Both the non-async and async version of the methods produce similar end results.



Email Tester in a Razor Page



Async Email Tester in a Razor Page

## Setting Attributes and Content

So how does the **Process()** method convert your custom tags into valid HTML tags? It does that in a series of steps.

1. Set the HTML element as the tag name to replace it with, e.g. `<a>`
2. Set each attribute within that HTML element, e.g. `href`
3. Set HTML Content *within* the tags.

The process involved is slightly different between the synchronous and asynchronous versions of the **Process** method. In the synchronous `EmailTagHelper.cs` class, the **Process()** method does the following:

```
// 1. Set the HTML element
output.TagName = "a";
```

```
// 2. Set the href attribute
output.Attributes.SetAttribute("href", "mailto:" + address);

// 3. Set HTML Content
output.Content.SetContent(address);
```

In the asynchronous `AsyncEmailTagHelper.cs` class, the `ProcessAsync()` method does the following:

```
// 1. Set the HTML element
output.TagName = "a";

var content = await output.GetChildContentAsync();
var target = content.GetContent() + "@" + EmailDomain;

// 2. Set the href attribute within that HTML element, e.g. href
output.Attributes.SetAttribute("href", "mailto:" + target);

// 3. Set HTML Content
output.Content.SetContent(target);
```

The difference between the two is that the `async` method gets the output content *asynchronously* with some additional steps. Before setting the attribute in Step 2, it grabs the output content from `GetChildContentAsync()` and then uses `content.GetContent()` to extract the content before setting the attribute with `output.Attributes.SetAttribute()`.

## Updating Pre/Post Content

This section recaps the **BoldTagHelper** explained in the docs tutorial, by consolidating all the lessons learned. In the `BoldTagHelper.cs` class from the sample, you can see the following code:

```
[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>");
    }
}
```

Let's go over what the code does, line by line:

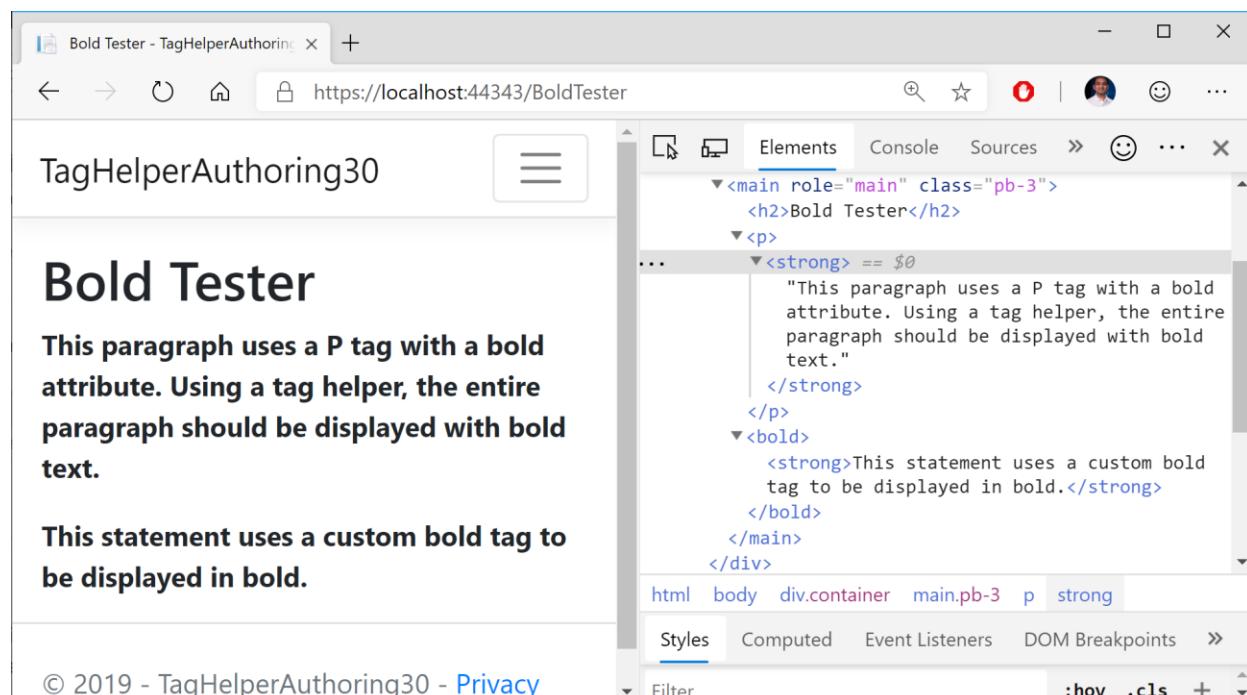
- The **[HtmlTargetElement]** attribute forces a Tag Helper to target a specific element, e.g. **[HtmlTargetElement("bold")]**, which will target a **<bold>** tag in a Razor Page or MVC View.
- When one or more attributes are specified, e.g. **[HtmlTargetElement(Attributes = "bold")]**, the Tag Helper targets a **boldattribute** within an element, e.g. **<p bold>**
- Combining the above one after the other gives you an OR condition, in which either scenario can be matched.
- Combining them in a single **[HtmlTargetElement]** creates an AND condition, which would match a bold tag with a bold attribute, which is not very useful, e.g. **[HtmlTargetElement("bold", Attributes = "bold")]**

Here is a snippet the corresponding Razor Page for testing the above scenario, BoldTester.cshtml:

```
<p bold>This paragraph uses a P tag with a bold attribute.  
Using a tag helper, the entire paragraph should be displayed with bold  
text.</p>
```

```
<bold>This statement uses a custom bold tag to be displayed in  
bold.</bold>
```

The tag helper affects both fragments, as seen in the screenshot below:



The statements in the **Process()** method accomplish the following:

- The **RemoveAll()** method from **output.Attributes** removes the “bold” attribute within the tag, as it is essentially acting as a placeholder.
- The **SetHtmlContent()** from **output.PreContent** adds an *opening* **<strong>** tag inside the enclosing element, i.e. just after **<p>** or **<bold>**
- The **SetHtmlContent()** from **output.PostContent** adds a *closing* **</strong>** tag inside the enclosing element, i.e. just before **</p>** or **</bold>**

## Passing Complex Objects

What if you want to pass a more complex object, with properties and objects within it? This can be done by defining a C# model class, e.g. **SuperheroModel.cs**, that can be initialized inside in the Page Model class (**SuperheroInfoTesterModel.cs**) and then used in a Razor Page (**SuperheroInfoTester.cshtml**). The tag helper (**SuperheroTagHelper.cs**) then brings it all together by replacing **<superhero>** tags with whatever **SuperHeroModel** info is passed in.

Let's take a look at all its parts, and how it all comes together.

### Object Model: SuperheroModel.cs

```
public class SuperheroModel
{
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public string SuperName { get; set; }
    public bool HasSurvived { get; set; }

    public bool ShowInfoWithSpoilers { get; set; }
}
```

### Razor Page: SuperheroInfoTester.cshtml

```
@page
@model SuperheroInfoTesterModel
```

```

...
<h3>Black Widow Info:</h3>
<div condition="@Model.blackWidowInfo.ShowInfoWithSpoilers">
  <superhero hero-info="Model.blackWidowInfo" />
</div>
...

```

#### **Page Model for Razor Page:** SuperheroInfoTester.cshtml.cs

```

public class SuperheroInfoTesterModel : PageModel
{
    public SuperheroModel blackWidowInfo { get; set; }
    // ...

    public void OnGet()
    {
        blackWidowInfo = new SuperheroModel
        {
            // ...
        }
        // ...
    }
}

```

#### **Superhero Tag Helper:** SuperheroTagHelper.cs

```

public class SuperheroTagHelper : TagHelper
{
    public SuperheroModel HeroInfo { get; set; }

    public override void Process(TagHelperContext context,
TagHelperOutput output)
    {
        // ...
    }
}

```

Going through the above code:

1. The tag helper is named **SuperheroTagHelper**, implying that it can be used for `<superhero>` tags in a Razor Page, e.g. SuperHeroInfoTester.cshtml
2. The tag helper also contains a **SuperheroModel** object called **HeroInfo**, which allows a hero-info attribute, i.e. `<superhero hero-info="Model.property">`
3. The **SuperheroModel** class contains various public properties that provide information about a specific superhero.

4. The **SuperHeroInfoTesterModel** page model class includes an **OnGet()** method that initializes multiple **SuperheroModel** objects, to be displayed in the Razor Page.

Inside the tag helper, the **Process()** method takes care of replacing the **<superhero>** tag with a **<section>** tag:

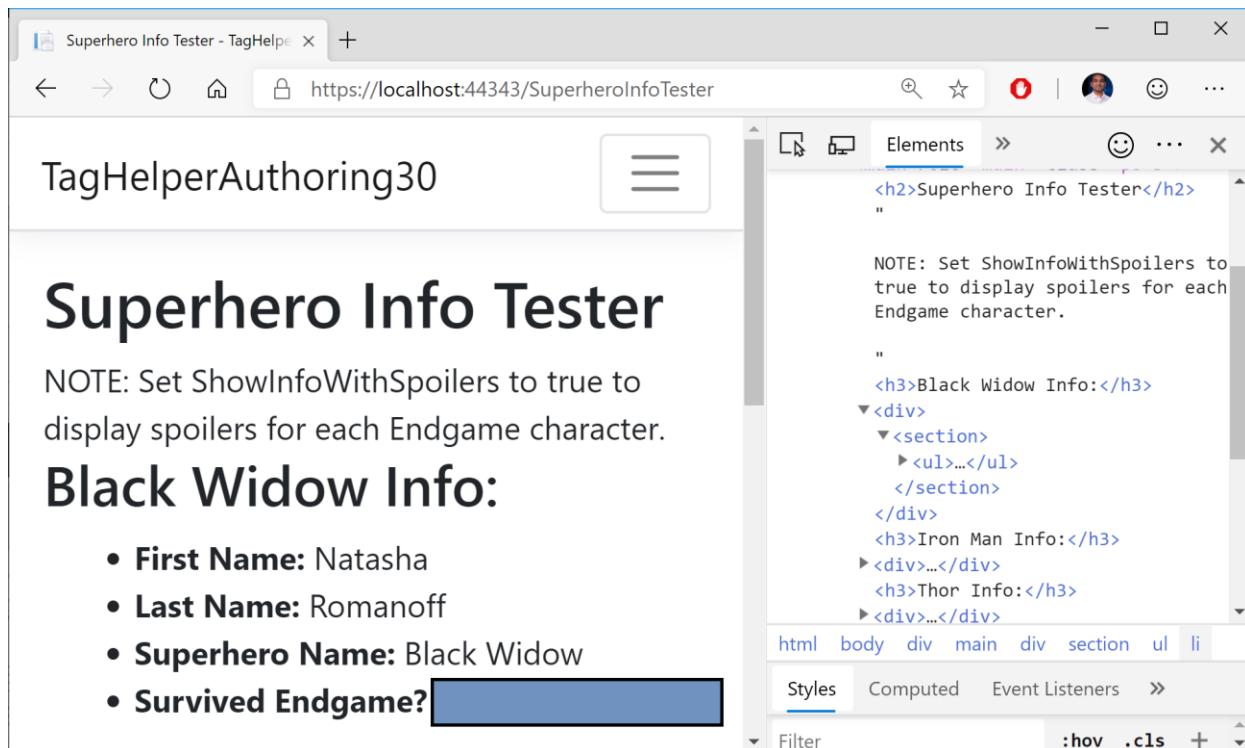
```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    string htmlContent = $@"
        <ul><li><strong>First Name:</strong>
        {HeroInfo.FirstName}</li>
        <li><strong>Last Name:</strong> {HeroInfo.LastName}</li>
        <li><strong>Superhero Name:</strong> {HeroInfo.SuperName}</li>
        <li><strong>Survived Endgame? </strong>
        {HeroInfo.HasSurvived}</li></ul>";

    output.TagName = "section";
    output.Content.SetHtmlContent(htmlContent);
    output.TagMode = TagMode.StartTagAndEndTag;
}
```

After initializing some HTML content to display a **<ul>** list, the above code in the **Process()** method accomplishes the following:

1. Set the HTML element as the tag name to replace it with, e.g. **<section>**
2. Set HTML Content *within* the tags.
3. Set Tag Mode to include both start and end tags, e.g. **<section> ... </section>**

**End Result in Browser:**



In a web browser, you can see that the `<superhero>` tag has been converted into a `<section>` tag with `<ul>` content.

## Handling Conditions

When you want to handle a UI element in different ways based on certain conditions, you may use a `ConditionTagHelper`. In this case, a condition is used to determine whether spoilers for the popular movie *Avengers: Endgame* should be displayed or not. If the spoiler flag is set to false, the character's info is not displayed at all.

```
@page
@model SuperheroInfoTesterModel
...
<div condition="@Model.blackWidowInfo.ShowInfoWithSpoilers">
  <superhero hero-info="Model.blackWidowInfo" />
</div>
...

```

In the above code from the `SuperheroInfoTester.cshtml` page:

- the <div> includes a condition that evaluates a boolean value, e.g. Model.blackWidowInfo.ShowInfoWithSpoilers
- the Model object comes from the @model defined at the top of the page
- the boolean value of **ShowInfoWithSpoilers** determines whether the <div> is displayed or not.

## References

- Tag Helpers in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro>
- Tag Helpers in forms in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms>
- Author Tag Helpers in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/authoring>

# Unit Testing in ASP .NET Core

By Shahed C on May 28, 2019

10 Replies

This is the **twenty-first** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



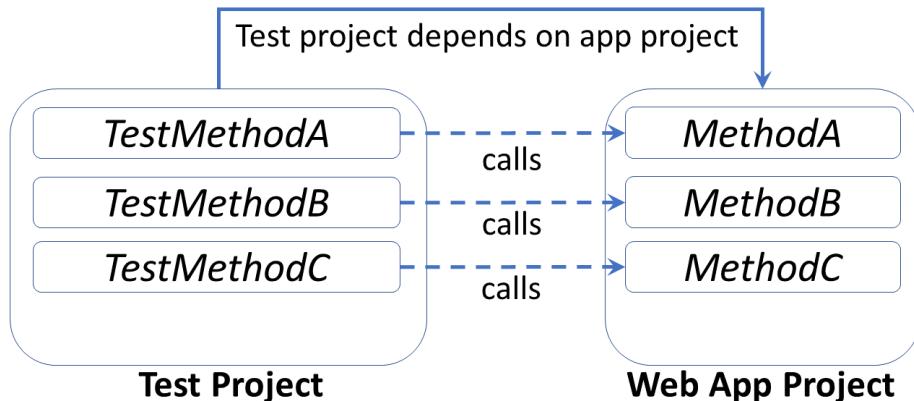
## In this Article:

- U is for Unit Testing
- Setting up Unit Testing
- Facts, Theories and Inline Data
- Asserts and Exceptions
- Running Tests
- Custom Names and Categories
- Next Steps
- References

# U is for Unit testing

Whether you're practicing TDD (Test-Driven Development) or writing your tests after your application code, there's no doubt that unit testing is essential for web application development. When it's time to pick a testing framework, there are multiple alternatives such as xUnit.net, NUnit and MSTest. This article will focus on xUnit.net because of its popularity (and *similarity* to its alternatives) when compared to the other testing frameworks.

In a nutshell: a unit test is code you can write to test your application code. Your web application will not have any knowledge of your test project, but your test project will need to have a dependency of the app project that it's testing.



Here are some poll results, from asking 500+ developers about which testing framework they prefer, showing xUnit.net in the lead.

POLL: Hey #AspNetCore #webdev community on twitter! What do you use for unit testing #ASPNET web apps?

Please RT for reach. Thanks!

— Shahed (on Leave) (@shahedC) May 22, 2019

A similar poll on Facebook also showed xUnit.net leading ahead of other testing frameworks. If you need to see the equivalent attributes and assertions, check out the comparison table provided by xUnit.net:

- Comparing xUnit.net to other frameworks: <https://xunit.net/docs/comparisons>

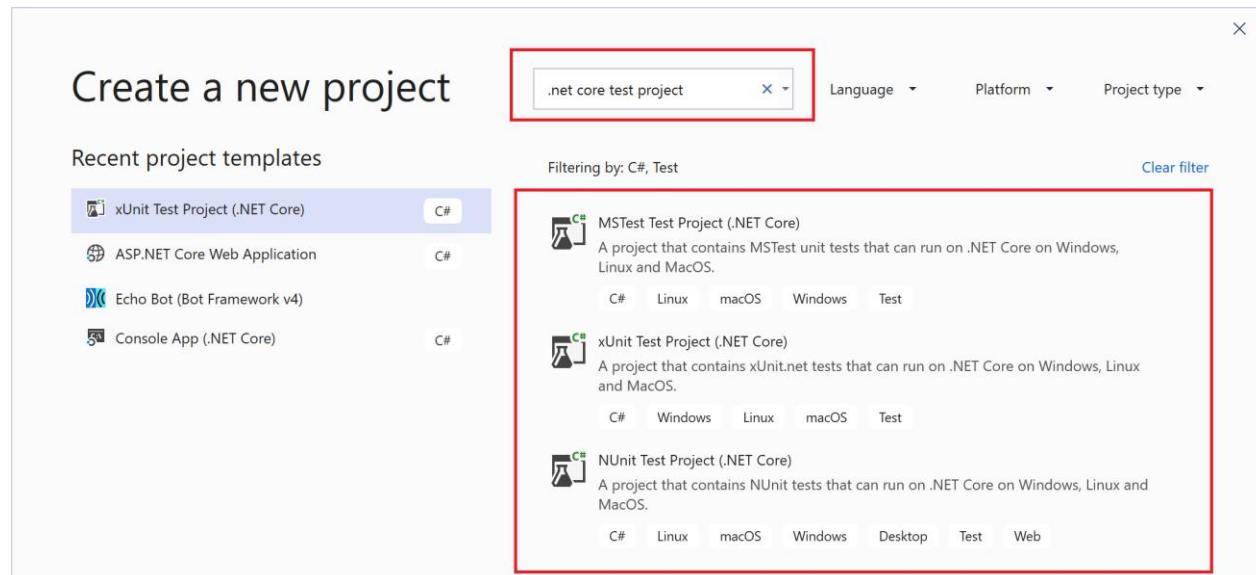
To follow along, take a look at the sample project on Github:



Unit Testing Sample: <https://github.com/shahedc/CalcApp30WithTests>

## Setting up Unit Testing

The quickest way to set up unit testing for an ASP .NET Core web app project is to create a new test project using a template. This creates a cross-platform .NET Core project that includes one blank test. In Visual Studio 2019, search for “.net core test project” when creating a new project to identify test projects for MSTest, XUnit and NUnit. Select the **XUnit** project to follow along with the sample provided.



The placeholder unit test class includes a blank test. Typically, you could create a test class for each application class being tested. The simplest unit test usually includes three distinct steps: Arrange, Act and Assert.

1. **Arrange:** Set up the any variables and objects necessary.
2. **Act:** Call the method being tested, passing any parameters needed
3. **Assert:** Verify expected results

The unit test project should have a dependency for the app project that it's testing. In the test project file CalcMvcWeb.Tests.csproj, you'll find a reference to CalcMvcWeb.csproj.

```
...
<ItemGroup>
    <ProjectReference Include="..\..\CalcMvcWeb\CalcMvcWeb.csproj" />
</ItemGroup>
...

```

In the Solution Explorer panel, you should see a project dependency of the reference project.



If you need help adding reference projects using CLI commands, check out the official docs at:

- Unit testing C# code in .NET Core using dotnet test and xUnit: <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test#creating-the-test-project>

# Facts, Theories and Inline Data

When you add a new xUnit test project, you should get a simple test class (**UnitTest1**) with an empty test method (**Test1**). This test class should be a public class and the test method should be decorated with a [**Fact**] attribute. The attribute indicates that this is a test method without any parameters, e.g. **Test1()**.

```
public class UnitTest1
{
    [Fact]
    public void Test1()
    {

    }
}
```

In the sample project, you'll see a test class (CalcServiceTests.cs) with a series of methods that take 1 or more parameters. Instead of a [**Fact**] attribute, each method has a [**Theory**] attribute. In addition to this primary attribute, each [**Theory**] attribute is followed by one or more [**InlineData**] attributes that have sample argument values for each method parameter.

```
[Theory(DisplayName = "Add Numbers")]
[InlineData(4, 5, 9)]
[InlineData(2, 3, 5)]
public void TestAddNumbers(int x, int y, int expectedResult)
{
    ...
}
```

In the code sample, each occurrence of [**InlineData**] is followed by 3 numeric values:

- [**InlineData(4, 5, 9)**] → this implies that x = 4, y = 5, expectedResult = 9
- [**InlineData(2, 3, 5)**] → this implies that x = 2, y = 3, expectedResult = 5

**NOTE:** If you want to skip a method during your test runs, simply add a **Skip** parameter to your **Fact** or **Theory** with a text string for the “Reason”.

e.g.

- [**Fact(Skip="this is broken")**]

- [Theory(Skip="we should skip this too")]

## Asserts and Exceptions

Back to the 3-step process, let's explore the **TestAddNumbers()** method and its method body.

```
public void TestAddNumbers(int x, int y, int expectedResult)
{
    // 1. Arrange
    var cs = new CalcService();

    // 2. Act
    var result = cs.AddNumbers(x, y);

    // 3. Assert
    Assert.Equal(expectedResult, result);
}
```

1. During the **Arrange** step, we create a new instance of an object called **CalcService** which will be tested.
2. During the **Act** step, we call the object's **AddNumbers()** method and pass 2 values that were passed in via **InlineData**.
3. During the **Assert** step, we compare the **expectedResult** (passed by **InlineData**) with the returned result (obtained from a call to the method being tested).

The **Assert.Equal()** method is a quick way to check whether an expected result is equal to a returned result. If they are equal, the test method will pass. Otherwise, the test will fail. There is also an **Assert.True()** method that can take in a boolean value, and will pass the test if the boolean value is true.

For a complete list of Assertions in xUnit.net, refer to the Assertions section of the aforementioned comparison table:

- Assertions in unit testing frameworks: <https://xunit.net/docs/comparisons#assertions>

If an exception is expected, you can *assert* a thrown exception. In this case, the test *passes* if the exception occurs. Keep in mind that unit tests are for testing *expected* scenarios. You can only test for an exception if you know that it will occur.

```
[Theory]
[InlineData(1, 0)]
public void TestDivideByZero(int x, int y)
{
    var cs = new CalcService();

    Exception ex = Assert
        .Throws<DivideByZeroException>(() => cs.UnsafeDivide(x, y));

}
```

The above code tests a method named **UnsafeDivide()** that divides 2 numbers, x and y. There is no guard against dividing by zero, so a **DivideByZeroException()** occurs whenever y is 0. Since **InlineData** passes a 1 and a 0, this guarantees that the exception will occur. In this case, the Act and Assert steps occur in the same statement.

To get a glimpse of the **UnsafeDivide()** method from the **CalcService** class, here is a snippet:

```
public int UnsafeDivide(int x, int y)
{
    return (x / y);
}
```

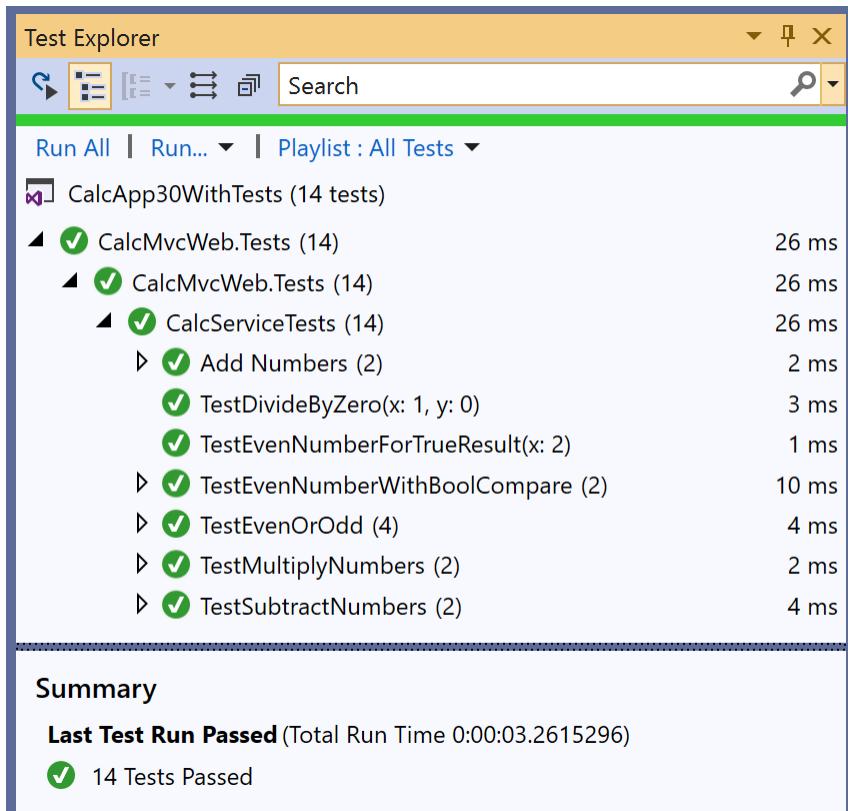
As you can see in the code snippet above, the **UnsafeDivide()** method simply divides two numbers, without checking to see if the denominator is 0 or not. This generates the expected exception when y is set to 0.

## Running Tests

To run your unit tests in Visual Studio, use the Test Explorer panel.

1. From the top menu, click Test | Windows | Test Explorer
2. In the Test Explorer panel, click Run All
3. Review the test status and summary

4. OPTIONAL: if any tests fail, inspect the code and fix as needed.



To run your unit tests with a CLI Command, run the following command in the test project folder:

```
> dotnet test
```

The results may look something like this:

```
Test run for
C:\path\to\test\assembly\CalcMvcWeb.Tests.dll(.NETCoreApp,Version=v3.0
)
Microsoft (R) Test Execution Command Line Tool Version 16.0.1
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
Starting test execution, please wait...
```

```
Total tests: 14. Passed: 14. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 2.4306 Seconds
```

As of xUnit version 2, tests can automatically run in parallel to save time. Test methods within a class are considered to be in the same implicit *collection*, and so will not be run in parallel. You can also

define *explicit* collections using a [Collection] attribute to decorate each test class. Multiple test classes within the same collection will not be run in parallel.

For more information on collections, check out the official docs at:

- Running Tests in Parallel: <https://xunit.net/docs/running-tests-in-parallel>

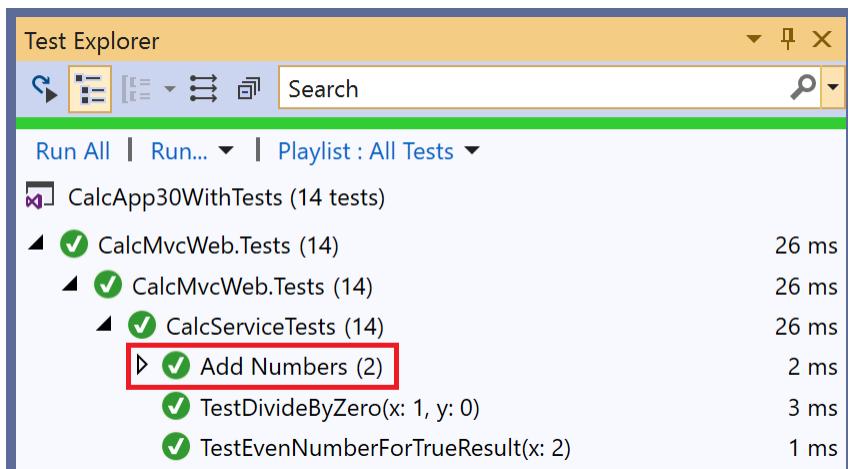
**NOTE:** Visual Studio includes a Live Unit Testing feature that allows you to see the status of passing/failing tests as you're typing your code. This feature is only available in the Enterprise Edition of Visual Studio.

## Custom Names and Categories

You may have noticed a **DisplayName** parameter when defining the [Theory] attribute in the code samples. This parameter allows you to define a friendly name for any test method (Fact or Theory) that can be displayed in the Test Explorer. For example:

```
[Theory(DisplayName = "Add Numbers")]
```

Using the above attribute above the **TestAddNumbers()** method will show the friendly name “**Add Numbers**” in the Test Explorer panel during test runs.

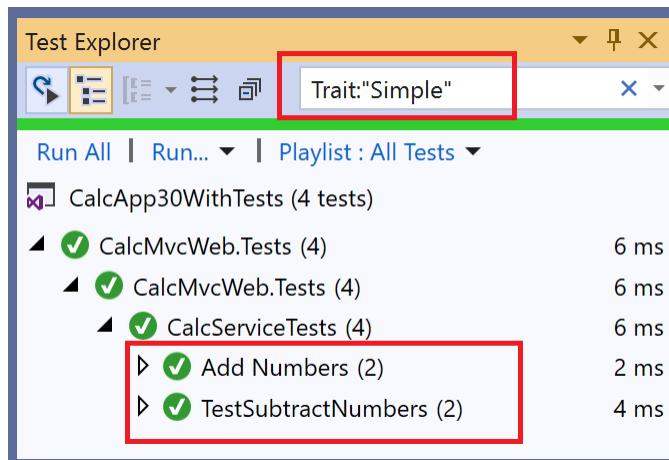


Finally, consider the **[Trait]** attribute. This attribute can be used to categorize related test methods by assigning an arbitrary name/value pair for each defined “Trait”. For example:

```
[Trait("Math Ops", "Simple")]
public void TestAddNumbers() { ... }
```

```
[Trait("Math Ops", "Simple")]
public void TestSubtractNumbers() { ... }
```

Using the above attribute for two methods **TestAddNumbers()** and **TestSubtractNumbers()** will categorize the methods into a “category” called Math Ops: Simple. This makes it possible to filter just the Addition and Subtraction test methods, e.g. Trait: “Simple”



## Next Steps: Mocking, Integration Tests and More!

There is so much more to learn with unit testing. You could read several chapters or even an entire book on unit testing and related topics. To continue your learning in this area, consider the following:

- **MemberData**: use the MemberData attribute to go beyond isolated test methods. This allows you to reuse test values for multiple methods in the test class.

- **ClassData:** use the ClassData attribute to use your test data in multiple test classes. This allows you to specify a class that will pass a set of collections to your test method.

For more information on the above, check out this Nov 2017 post from Andrew Lock:

- Creating parameterised tests in xUnit with [InlineData], [ClassData], and [MemberData]:  
<https://andrewlock.net/creating-parameterised-tests-in-xunit-with-inlinedata-classdata-and-memberdata/>

To go beyond Unit Tests, consider the following:

- **Mocking:** use a mocking framework (e.g. Moq) to mock external dependencies that you shouldn't need to test from your own code.
- **Integration Tests:** use integration tests to go beyond isolated unit tests, to ensure that multiple components of your application are working correctly. This includes databases and file systems.
- **UI Tests:** test your UI components using a tool such as Selenium WebDriver or IDE in the language of your choice, e.g. C#. For browser support, you may use Chrome or Firefox extensions, so this includes the new Chromium-based Edge browser.

While this article only covers MVC, take a look at the official docs for Unit Testing with Razor Pages:

- Razor Pages unit tests in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/test/razor-pages-tests>

## References

- Getting started: .NET Core with command line > xUnit.net: <https://xunit.net/docs/getting-started/netcore/cmdline>
- Running Tests in Parallel > xUnit.net: <https://xunit.net/docs/running-tests-in-parallel>

- Unit testing C# code in .NET Core using dotnet test and xUnit: <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test>
- Test controller logic in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/testing>
- Integration tests in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests>
- Razor Pages unit tests in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/test/razor-pages-tests>
- Unit testing in .NET Core and .NET Standard – .NET Core: <https://docs.microsoft.com/en-us/dotnet/core/testing/>
- Introducing the .NET Core Unit Testing Framework (or: Why xUnit?): <https://visualstudiomagazine.com/articles/2018/11/01/net-core-testing.aspx?m=1>
- Writing xUnit Tests in .NET Core: <https://visualstudiomagazine.com/articles/2018/11/01/xunit-tests-in-net-core.aspx>
- Live Unit Testing – Visual Studio: <https://docs.microsoft.com/en-us/visualstudio/test/live-unit-testing>
- Mocks Aren't Stubs: <https://martinfowler.com/articles/mockArentStubs.html>
- Mocking in .NET Core Tests with Moq: <http://dontcodetired.com/blog/post/Mocking-in-NET-Core-Tests-with-Moq>
- Moq – Unit Test In .NET Core App Using Mock Object: <https://www.c-sharpcorner.com/article/moq-unit-test-net-core-app-using-mock-object/>

# Validation in ASP .NET Core

By Shahed C on June 4, 2019

6 Replies

This is the **twenty-second** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



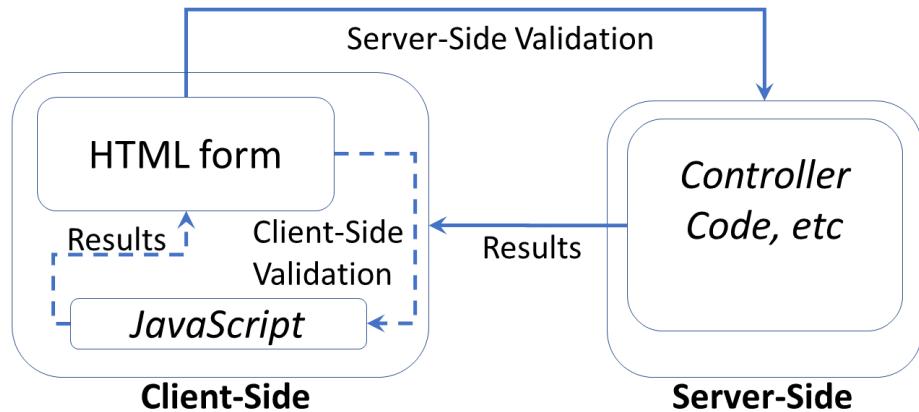
## In this Article:

- V is for Validation
- Validation Attributes
- Server-Side Validation
- Client-Side Validation
- Client to Server with Remote Validation
- Custom Attributes
- References

# V is for Validation

To build upon a previous post on Forms and Fields in ASP .NET Core, this post covers Validation in ASP .NET Core. When a user submits form field values, proper validation can help build a more user-friendly and secure web application. Instead of coding each view/page individually, you can simply use server-side attributes in your models/viewmodels.

**NOTE:** As of ASP .NET Core 2.2, validation may be skipped automatically if ASP .NET Core decides that validation is not needed. According to the “What’s New” release notes, this includes primitive collections (e.g. a byte[] array or a Dictionary<string, string> key-value pair collection)



This article will refer to the following sample code on GitHub:



Validation Sample App: <https://github.com/shahedc/ValidationSampleApp>

## Validation Attributes

To implement model validation with [Attributes], you will typically use Data Annotations from the System.ComponentModel.DataAnnotations namespace. The list of attribute does go beyond just validation functionality though. For example, the DataType attribute takes a datatype parameter, used for inferring the data type and used for displaying the field on a view/page (but does not provide validation for the field).

Common attributes include the following

- **Range:** lets you specify min-max values, inclusive of min and max
- **RegularExpression:** useful for pattern recognition, e.g. phone numbers, zip/postal codes
- **Required:** indicates that a field is required
- **StringLength:** sets the maximum length for the string entered
- **MinLength:** sets the minimum length of an array or string data

From the sample code, here is an example from the CinematicItem model class:

```
public class CinematicItem
{
    public int Id { get; set; }

    [Range(1,100)]
    public int Score { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [StringLength(255)]
    public string Synopsis { get; set; }

    [DataType(DataType.Date)]
    [DisplayName("Available Date")]
    public DateTime AvailableDate { get; set; }

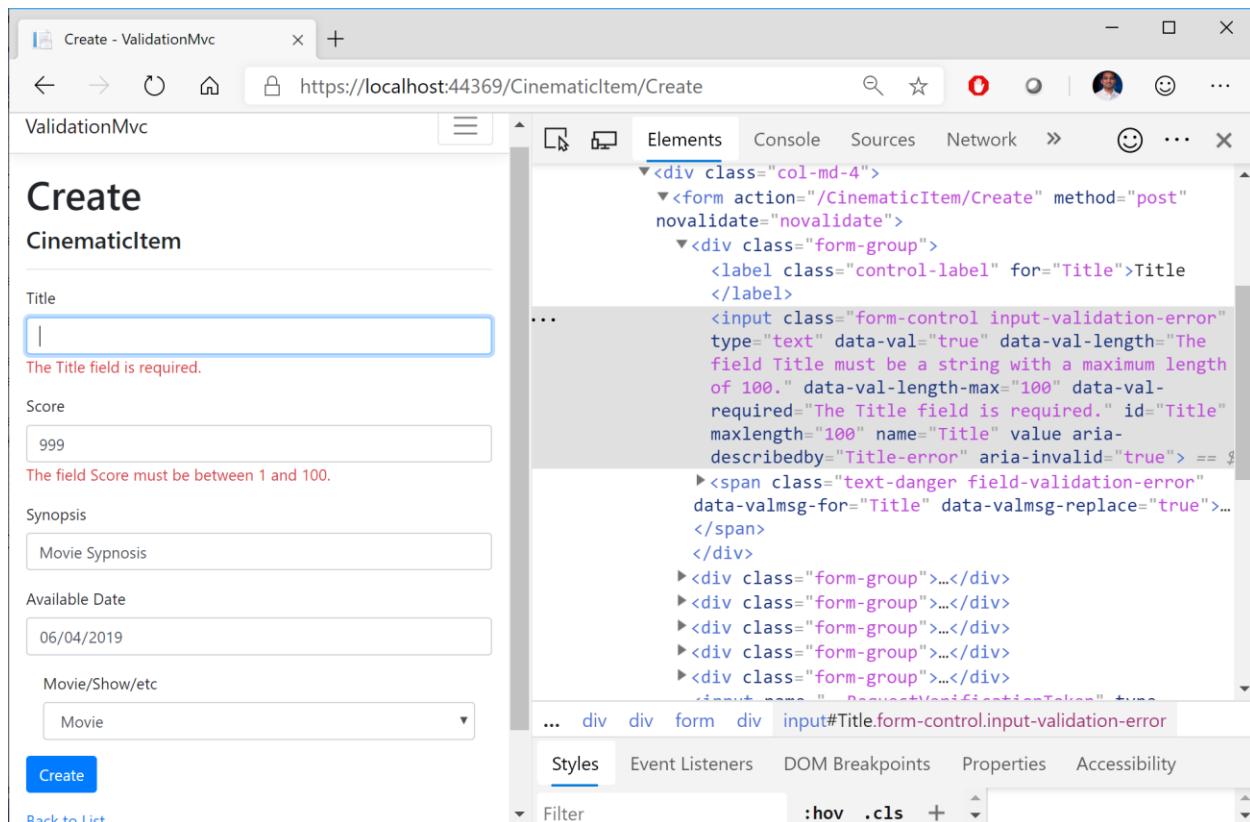
    [Required]
    [DisplayName("Movie/Show/etc")]
    public CIType CIType { get; set; }
}
```

From the above code, you can see that:

- The value for **Score** can be 1 or 100 or any integer in between
- The value for **Title** is a required string, needs to be less than 100 characters
- The value for **Synopsis** can be left blank, but has to be less than 100 characters.
- The value for **AvailableDate** is displayed as “Available Date” (with a space)
- Because of the **DataType** provided, **AvailableDate** is displayed as a selectable date in the browser
- The value for **CIType** (short for Cinematic Item Type) is displayed as “Movie>Show/etc” and is displayed as a selectable value obtained from the CIType data type (which happens to be an enumerator. (shown below)

```
public enum CIType
{
    Movie,
    Series,
    Short
}
```

Here's what it looks like in a browser when validation fails:



The validation rules make it easier for the user to correct their entries before submitting the form.

## Server-Side Validation

Validation occurs before an MVC controller action (or equivalent handler method for Razor Pages) takes over. As a result, you should check to see if the validation has passed before continuing next steps.

e.g. in an MVC controller

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(....)
{
    if (ModelState.IsValid)
    {
        // ...
        return RedirectToAction(nameof(Index));
    }
    return View(cinematicItem);
}
```

e.g. in a Razor Page's handler code:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    //...
    return RedirectToPage("./Index");
}
```

Note that **ModelState.IsValid** is checked in both the **Create()** action method of an MVC Controller or the **OnPostAsync()** handler method of a Razor Page's handler code. If **IsValid** is true, perform actions as desired. If false, reload the current view/page as is.

# Client-Side Validation

It goes without saying that you should always have server-side validation. All the client-side validation in the world won't prevent a malicious user from sending a GET/POST request to your form's endpoint. Cross-site request forgery in the Form tag helper does provide a certain level of protection, but you still need server-side validation. That being said, client-side validation helps to catch the problem before your server receives the request, while providing a better user experience.

When you create a new ASP .NET Core project using one of the built-in templates, you should see a shared partial view called `_ValidationScriptsPartial.cshtml`. This partial view should include references to jQuery unobtrusive validation, as shown below:

```
<environment include="Development">
    <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
    <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>
</environment>
```

If you create a scaffolded controller with views/pages, you should see the following reference at the bottom of your page or view.

e.g. at the bottom of Create.cshtml view

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

e.g. at the bottom of the Create.cshtml page

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

Note that the syntax is identical whether it's an MVC view or a Razor page. That being said, you may want to disable client-side validation. This is accomplished in different ways, whether it's for an MVC view or a Razor page.

From the official docs, the following code should be used within the `ConfigureServices()` method of your `Startup.cs` class, to set `ClientValidationEnabled` to false in your `HTMLHelperOptions` configuration.

```
services.AddMvc().AddViewOptions(options =>
{
```

```
if (_env.IsDevelopment())
{
    options.HtmlHelperOptions.ClientValidationEnabled = false;
}
});
```

Also mentioned in the official docs, the following code can be used for your Razor Pages, within the **ConfigureServices()** method of your Startup.cs class.

```
services.Configure<HtmlHelperOptions>(o => o.ClientValidationEnabled = false);
```

## Client to Server with Remote Validation

If you need to call a server-side method while performing client-side validation, you can use the **[Remote]** attribute on a model property. You would then pass it the name of a server-side action method which returns an **IActionResult** with a true boolean result for a valid field. This **[Remote]** attribute is available in the Microsoft.AspNetCore.Mvc namespace, from the Microsoft.AspNetCore.Mvc.ViewFeatures NuGet package.

The model property would look something like this:

```
[Remote(action: "MyActionMethod", controller: "MyControllerName")]
public string MyProperty { get; set; }
```

In the controller class, (e.g. **MyControllerName**), you would define an action method with the name specified in the **[Remote]** attribute parameters, e.g. **MyActionMethod**.

```
[AcceptVerbs("Get", "Post")]
public IActionResult MyActionMethod(...)
{
    if (TestForFailureHere())
    {
        return Json("Invalid Error Message");
    }
    return Json(true);
}
```

You may notice that if the validation fails, the controller action method returns a JSON response with an appropriate error message in a string. Instead of a text string, you can also use a false, null, or undefined value to indicate an invalid result. If validation has passed, you would use **Json(true)** to indicate that the validation has passed.

*So, when would you actually use something like this?* Any scenario where a selection/entry needs to be validated by the server can provide a better user experience by providing a result as the user is typing, instead of waiting for a form submission. For example: imagine that a user is buying online tickets for an event, and selecting a seat number displayed on a seating chart. The selected seat could then be displayed in an input field and then sent back to the server to determine whether the seat is still available or not.

## Custom Attributes

In addition to all of the above, you can simply build your own custom attributes. If you take a look at the classes for the built-in attributes, e.g. RequiredAttribute, you will notice that they also extend the same parent class:

- System.ComponentModel.DataAnnotations.ValidationAttribute

You can do the same thing with your custom attribute's class definition:

```
public class MyCustomAttribute: ValidationAttribute
{
    // ...
}
```

The parent class ValidationAttribute, has a virtual **IsValid()** method that you can override to return whether validation has been calculated successfully (or not).

```
public class MyCustomAttribute: ValidationAttribute
{
    // ...
    protected override ValidationResult IsValid(
        object value, ValidationContext validationContext)
    {
        if (TestForFailureHere())
        {
            return new ValidationResult("Invalid Error Message");
        }

        return ValidationResult.Success;
    }
}
```

You may notice that if the validation fails, the **IsValid()** method returns a **ValidationResult()** with an appropriate error message in a string. If validation has passed, you would return **ValidationResult.Success** to indicate that the validation has passed.

## References

- Add validation to an ASP.NET Core MVC app: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation>
- Model validation in ASP.NET Core MVC and Razor Pages: <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation>
- System.ComponentModel.DataAnnotations Namespace: <https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations>
- ValidationAttribute Class  
(System.ComponentModel.DataAnnotations): <https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations.validationattribute>

# Worker Service in ASP .NET Core

By Shahed C on June 10, 2019

3 Replies

This is the **twenty-third** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



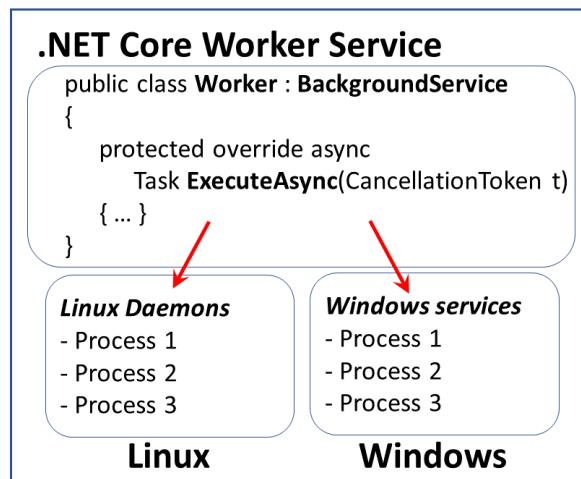
## In this Article:

- W is for Worker Service
- New Worker Service Project
- Program and BackgroundService
- Logging in a Worker Service
- Running the Worker Service
- References

## W is for Worker Service

When you think of ASP .NET Core, you probably think of web application backend code, including MVC and Web API. MVC Views and Razor Pages also allow you to use backend code to generate frontend UI with HTML elements. The all-new Blazor goes one step further to allow client-side .NET code to run in a web browser, using WebAssembly. And finally, we now have a template for Worker Service applications.

Briefly mentioned in a previous post in this series, the new project type was introduced in ASP .NET Core early previews. Although the project template is currently listed under the Web templates, it is expected to be relocated one level up in the New Project Wizard. This is a great way to create potentially long-running cross-platform services in .NET Core. This article covers the Windows operating system.



This article will refer to the following sample code on GitHub:

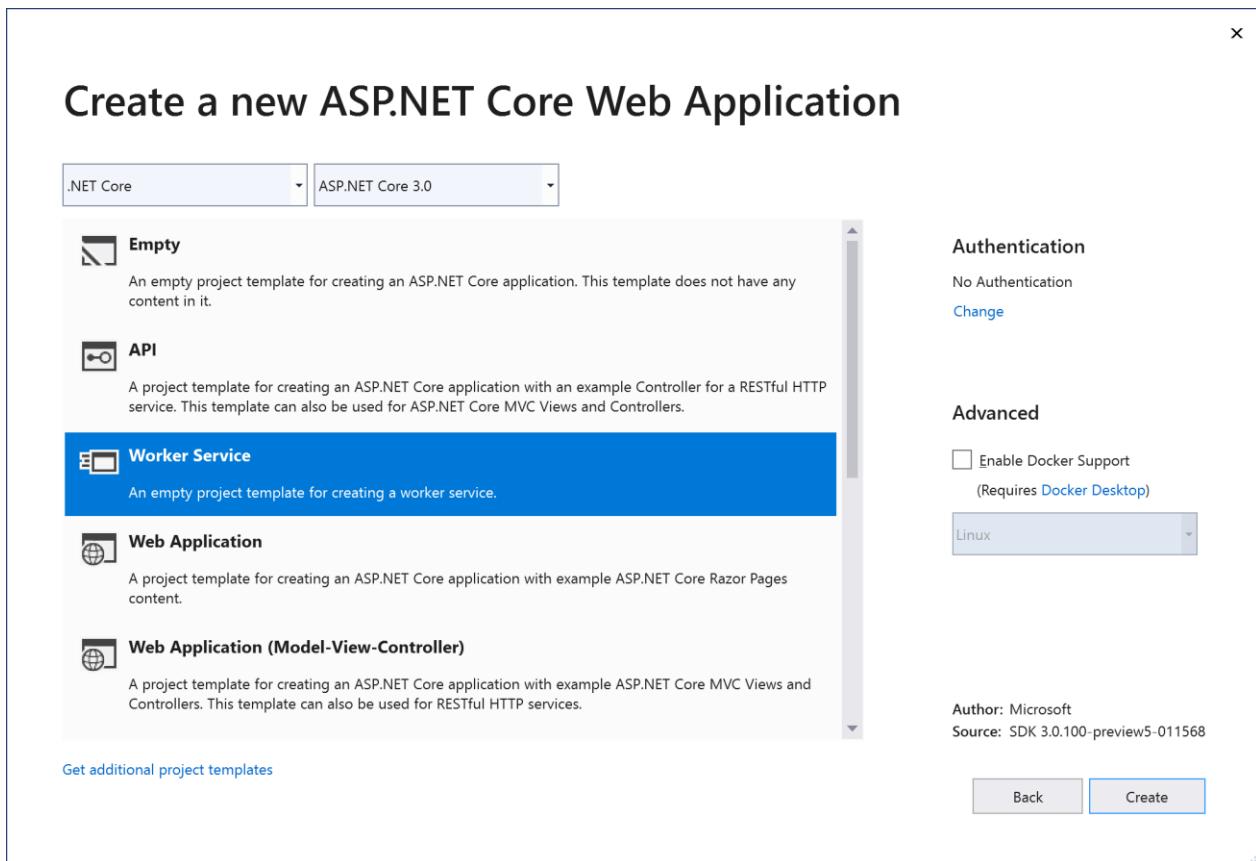


Worker Service Sample: <https://github.com/shahedc/WorkerServiceSample>

## New Worker Service Project

The quickest way to create a new Worker Service project in Visual Studio 2019 is to use the latest template available for ASP .NET Core 3.0. You may also use the appropriate dotnet CLI command.

Launch Visual Studio and select the Worker service template as shown below:



To use the Command Line, simply use the following command:

```
> dotnet new worker -o myproject
```

where `-o` is an optional flag to provide the output folder name for the project.

You can learn more about the new template at the following location:

- `dotnet worker`  
template: <https://dotnetnew.azurewebsites.net/template/Microsoft.DotNet.Web.ProjectTemplates.3.0/Microsoft.Worker.Empty.CSharp.3.0>

# Program and BackgroundService

The Program.cs class contains the usual **Main()** method and a familiar **CreateHostBuilder()** method. This can be seen in the snippet below:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .UseWindowsService()
            .ConfigureServices(services =>
            {
                services.AddHostedService<Worker>();
            });
}
```

Things to note:

1. The Main method calls the **CreateHostBuilder()** method with any passed parameters, builds it and runs it.
2. As of ASP .NET Core 3.0, the *Web* Host Builder is being replaced by a *Generic* Host Builder. The so-called Generic Host Builder was covered in an earlier blog post in this series.
3. **CreateHostBuilder()** creates the host and configures it by calling **AddHostService<T>**, where T is an **IHostedService**, e.g. a worker class that is a child of **BackgroundService**

The worker class, Worker.cs, is defined as shown below:

```
public class Worker : BackgroundService
{
    // ...

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        // do stuff here
    }
}
```

Things to note:

1. The worker class implements the **BackgroundService** class, which comes from the namespace Microsoft.Extensions.Hosting
2. The worker class can then override the **ExecuteAsync()** method to perform any long-running tasks.

In the sample project, a utility class (DocMaker.cs) is used to convert a web page (e.g. a blog post or article) into a Word document for offline viewing. Fun fact: when this A-Z series wraps up, the blog posts will be assembled into a free ebook, by using this EbookMaker, which uses some 3rd-party NuGet packages to generate the Word document.

## Logging in a Worker Service

Logging in ASP .NET Core has been covered in great detail in an earlier blog post in this series. To get a recap, take a look at the following writeup:

- Logging in ASP .NET Core: <https://wakeupandcode.com/logging-in-asp-net-core/>

To use Logging in your Worker Service project, you may use the following code in your Program.cs class:

```
using Microsoft.Extensions.Logging; public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseWindowsService()
        .ConfigureLogging(loggerFactory => loggerFactory.AddEventLog())
        .ConfigureServices(services =>
    {
    services.AddHostedService<Worker>();
}) ;
```

1. Before using the extension method, add its NuGet package to your project:
  - Microsoft.Extensions.Logging.EventLog
2. Add the appropriate namespace to your code:

- using Microsoft.Extensions.Logging;
3. Call the method **ConfigureLogging()** and call the appropriate logging method, e.g. **AddEventLog()**

The list of available loggers include:

- AddConsole()
- AddDebug()
- AddEventLog()
- AddEventSourceLogger()

The Worker class can then accept an injected **ILogger<Worker>** object in its constructor:

```
private readonly ILogger<Worker> _logger;

public Worker(ILogger<Worker> logger)
{
    _logger = logger;
}
```

## Running the Worker Service

**NOTE:** Run Powershell in Administrator Mode before running the commands below.

Before you continue, add a call to **UseWindowsService()** in your Program class, or verify that it's already there. The official announcement and initial document referred to **UseServiceBaseLifetime()** in an earlier preview. This method has been renamed to **UseWindowsService()** in the most recent version.

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseWindowsService()
        .ConfigureServices(services =>
    {
        services.AddHostedService<Worker>();
    });
}
```

According to the code documentation, **UseWindowsService()** does the following:

1. Sets the host lifetime to WindowsServiceLifetime
2. Sets the Content Root
3. Enables logging to the event log with the application name as the default source name

You can run the Worker Service in various ways:

1. Build and Debug/Run from within Visual Studio.
2. Publish to an exe file and run it
3. Run the sc utility (from Windows\System32) to create a new service

To publish the Worker Service as an exe file with dependencies, run the following **dotnet** command:

```
dotnet publish -o C:\path\to\project\pubfolder
```

The -o parameter can be used to specify the path to a folder where you wish to generate the published files. It could be the path to your project folder, followed by a new subfolder name to hold your published files, e.g. pubfolder. Make a note of your EXE name, e.g. MyProjectName.exe but omit the pubfolder from your source control system.

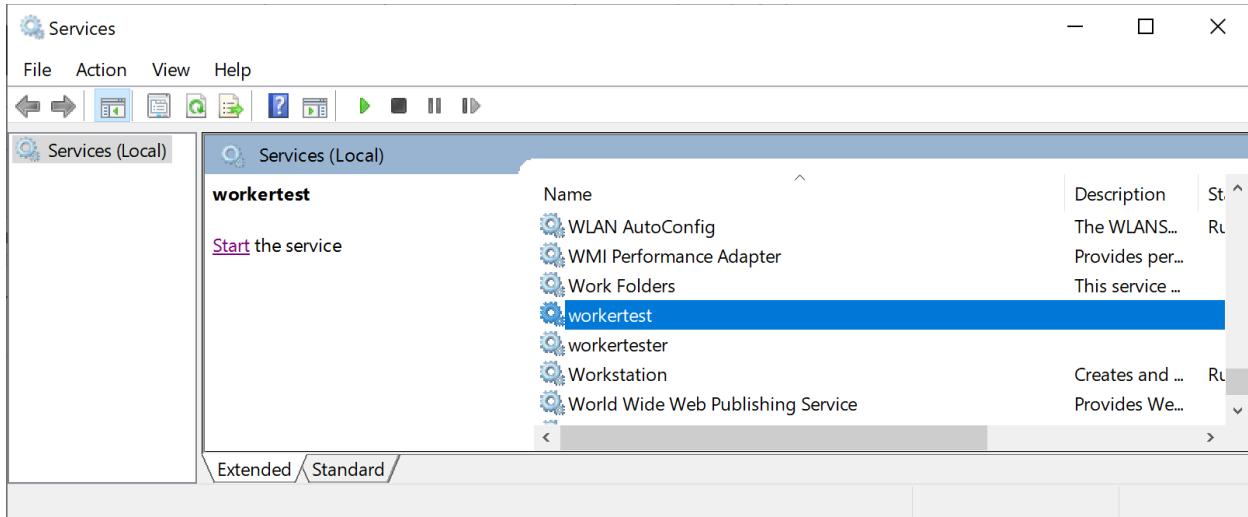
To create a new service, run **sc.exe** from your System32 folder and pass in the name of the EXE file generated from the publish command.

```
> C:\Windows\System32\sc create MyServiceName  
binPath=C:\path\to\project\pubfolder\MyProjectName.exe
```

When running the service manually, you should see some logging messages, as shown below:

```
info: WorkerServiceSample.Worker[0]  
Making doc 1 at: 06/09/2019 00:09:52 -04:00  
Making your document...  
info: WorkerServiceSample.Worker[0]  
Making doc 2 at: 06/09/2019 00:10:05 -04:00  
Making your document...  
info: Microsoft.Hosting.Lifetime[0]  
Application started. Press Ctrl+C to shut down.  
info: Microsoft.Hosting.Lifetime[0]  
Hosting environment: Development
```

After the service is installed, it should show up in the operating system's list of Windows Services:



**NOTE:** When porting to other operating systems, the call to **UseWindowsService()** is safe to leave as is. It doesn't do anything on a non-Windows system.

## References

- .NET Core Workers as Windows Services: <https://devblogs.microsoft.com/aspnet/net-core-workers-as-windows-services/>
- Worker Service template in .NET Core 3.0: <https://gunnarpeipman.com/net/worker-service/>
- Worker Service Template in .NET Core 3.0 – DZone Web Dev: <https://dzone.com/articles/worker-service-template-in-net-core-30>
- ASP.NET Blog | .NET Core Workers in Azure Container Instances: <https://devblogs.microsoft.com/aspnet/dotnet-core-workers-in-azure-container-instances/>

# XML + JSON Serialization in ASP .NET Core

By Shahed C on June 17, 2019

2 Replies

This is the **twenty-fourth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**

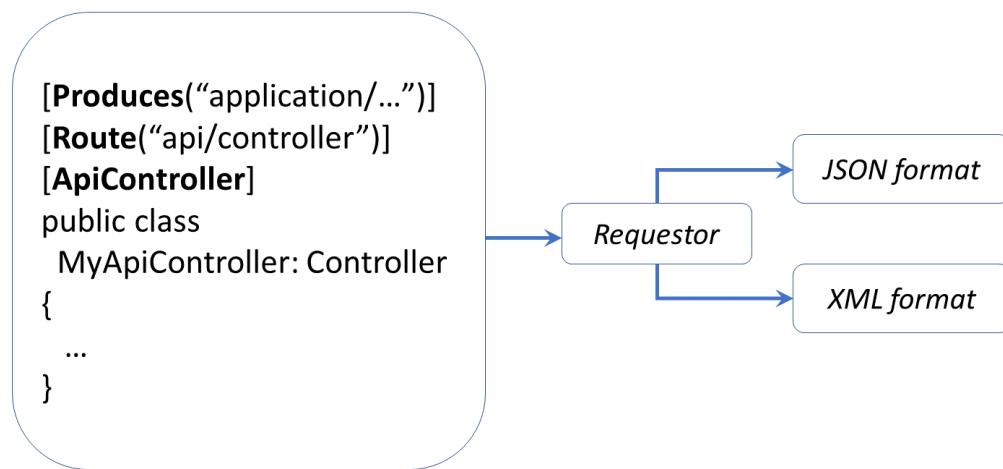


## In this Article:

- X is for XML + JSON Serialization
- Returning JsonResult and IActionResult
- Returning Complex Objects
- XML Serialization
- References

## X is for XML + JSON Serialization

XML (eXtensible Markup Language) is a popular document format that has been used for a variety of applications over the years, including Microsoft Office documents, SOAP Web Services, application configuration and more. JSON (JavaScript Object Notation) was derived from JavaScript, but has also been used for storing data in both structured and unstructured formats, regardless of language used. In fact, ASP .NET Core applications switched from XML-based .config files to JSON-based .json settings files for application configuration.



This article will refer to the following sample code on GitHub, derived from the guidance provided in the official documentation + sample:



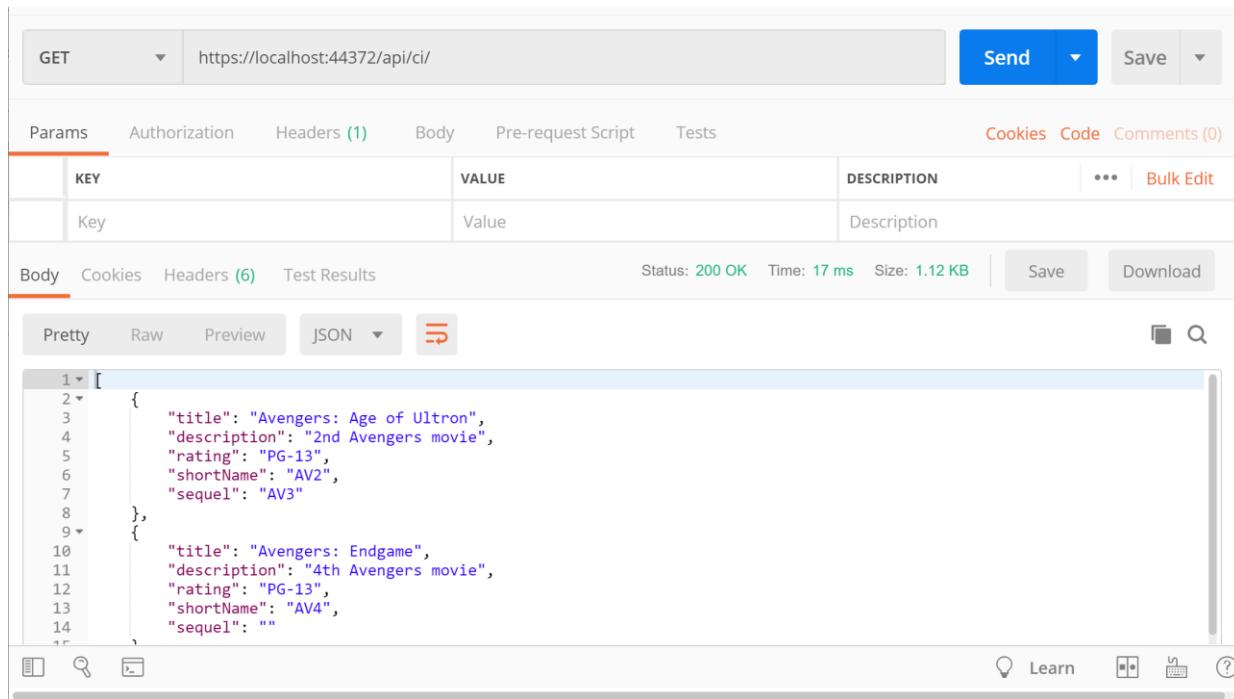
XML + JSON Serialization : <https://github.com/shahedc/XMLJsonSerialization>

## Returning JsonResult and IActionResult

Before we get into XML serialization, let's start off with JSON Serialization first, and then we'll get to XML. If you run the Web API sample project for this blog post, you'll notice a `CIController.cs` file that represents a "Cinematic Item Controller" that expose API endpoints. These endpoints can serve up both JSON and XML results of Cinematic Items, i.e. movies, shows and shorts in a Cinematic Universe.

Run the application and navigate to the following endpoint in an API testing tool, e.g. Postman:

- <https://localhost:44372/api/ci/>



The screenshot shows a Postman interface with a GET request to <https://localhost:44372/api/ci/>. The response body is a JSON array containing two movie objects:

```
[{"title": "Avengers: Age of Ultron", "description": "2nd Avengers movie", "rating": "PG-13", "shortName": "AV2", "sequel": "AV3"}, {"title": "Avengers: Endgame", "description": "4th Avengers movie", "rating": "PG-13", "shortName": "AV4", "sequel": ""}]
```

This triggers a GET request by calling the CIController's **Get()** method:

```
// GET: api/ci
[HttpGet]
public JsonResult Get()
{
    return Json(_cinematicItemRepository.CinematicItems());
}
```

In this case, the **Json()** method returns a **JsonResult** object that serializes a list of Cinematic Items. For simplicity, the `_cinematicRepository` object's `CinematicItems()` method (in `CinematicItemRepository.cs`) returns a hard-coded list of **CinematicItem** objects. Its implementation here isn't important, because you would typically retrieve such values from a persistent data store, preferably through some sort of service class.

```
public List<CinematicItem> CinematicItems()
{
    return new List<CinematicItem>
    {
        new CinematicItem
```

```

    {
        Title = "Iron Man 1",
        Description = "First movie to kick off the MCU.",
        Rating = "PG-13",
        ShortName = "IM1",
        Sequel = "IM2"
    },
    ...
}
}

```

The JSON result looks like the following, where a list of movies are returned:

```
[
{
    "title": "Avengers: Age of Ultron",
    "description": "2nd Avengers movie",
    "rating": "PG-13",
    "shortName": "AV2",
    "sequel": "AV3"
},
{
    "title": "Avengers: Endgame",
    "description": "4th Avengers movie",
    "rating": "PG-13",
    "shortName": "AV4",
    "sequel": ""
},
{
    "title": "Avengers: Infinity War",
    "description": "3rd Avengers movie",
    "rating": "PG-13",
    "shortName": "AV3",
    "sequel": "AV4"
},
{
    "title": "Iron Man 1",
    "description": "First movie to kick off the MCU.",
    "rating": "PG-13",
    "shortName": "IM1",
    "sequel": "IM2"
},
{
    "title": "Iron Man 2",
    "description": "Sequel to the first Iron Man movie.",
    "rating": "PG-13",
    "shortName": "IM2",
    "sequel": "IM3"
}
```

```

},
{
"title": "Iron Man 3",
"description": "Wraps up the Iron Man trilogy.",
"rating": "PG-13",
"shortName": "IM3",
"sequel": ""
},
{
"title": "The Avengers",
"description": "End of MCU Phase 1",
"rating": "PG-13",
"shortName": "AV1",
"sequel": "AV2"
}
]

```

Instead of specifically returning a **JsonResult**, you could also return a more generic **IActionResult**, which can still be interpreted as JSON. Run the application and navigate to the following endpoint, to include the action method “search” followed by a QueryString parameter “fragment” for a partial match.

- <https://localhost:44372/api/ci/search?fragment=ir>

KEY	VALUE	DESCRIPTION
fragment	ir	
Key	Value	Description

```

1 [
2   {
3     "title": "Iron Man 1",
4     "description": "First movie to kick off the MCU.",
5     "rating": "PG-13",
6     "shortName": "IM1",
7     "sequel": "IM2"
8   },
9   {
10    "title": "Iron Man 2",
11    "description": "Sequel to the first Iron Man movie.",
12    "rating": "PG-13",
13  }
14 ]

```

This triggers a GET request by calling the CIController's **Search()** method, with its fragment parameter set to "ir" for a partial text search:

```
// GET: api/ci/search?fragment=ir
[HttpGet("Search")]
public IActionResult Search(string fragment)
{
    var result = _cinematicItemRepository.GetByPartialName(fragment);
    if (!result.Any())
    {
        return NotFound(fragment);
    }
    return Ok(result);
}
```

In this case, the **GetByPartialName()** method returns a **List** of **CinematicItem** objects that are returned as JSON by default, with a HTTP 200 OK status. In case no results are found, the action method will return a 404 with the **NotFound()** method.

```
public List<CinematicItem> GetByPartialName(string titleFragment)
{
    return CinematicItems()
        .Where(ci => ci.Title
            .IndexOf(titleFragment, 0,
StringComparison.CurrentCultureIgnoreCase) != -1)
        .ToList();
}
```

The JSON result looks like the following, where any movie title partially matches the string fragment provided:

```
[
{
    "title": "Iron Man 1",
    "description": "First movie to kick off the MCU.",
    "rating": "PG-13",
    "shortName": "IM1",
    "sequel": "IM2"
},
{
    "title": "Iron Man 2",
    "description": "Sequel to the first Iron Man movie.",
    "rating": "PG-13",
    "shortName": "IM2",
    "sequel": "IM3"
},
{
    "title": "Iron Man 3",
    "description": "Wraps up the Iron Man trilogy.",
```

```
"rating": "PG-13",
"shortName": "IM3",
"sequel": ""
}
]
```

## Returning Complex Objects

An overloaded version of the **Get()** method takes in a “**shortName**” string parameter to filter results by an alternate short name for each movie in the repository for the cinematic universe. Instead of returning a **JsonResult** or **IActionResult**, this one returns a complex object (**CinematicItem**) that contains properties that we’re interested in.

```
// GET api/ci/IM1
[HttpGet("{shortName}")]
public CinematicItem Get(string shortName)
{
    return _cinematicItemRepository.GetByShortName(shortName);
}
```

The **GetByShortName()** method in the **CinematicItemRepository.cs** class simply checks for a movie by the **shortName** parameter and returns the first match. Again, the implementation is not particularly important, but it illustrates how you can pass in parameters to get back JSON results.

```
public CinematicItem GetByShortName(string shortName)
{
    return CinematicItems().FirstOrDefault(ci => ci.ShortName ==
shortName);
}
```

While the application is running, navigate to the following endpoint:

- <https://localhost:44372/api/ci/IM1>

The screenshot shows a Postman interface with a GET request to `https://localhost:44372/api/ci/IM1`. The response body is a JSON object:

```

1 {  
2   "title": "Iron Man 1",  
3   "description": "First movie to kick off the MCU.",  
4   "rating": "PG-13",  
5   "shortName": "IM1",  
6   "sequel": "IM2"  
7 }

```

This triggers another GET request by calling the `CiController`'s overloaded `Get()` method, with the `shortName` parameter. When passing the short name "IM1", this returns one item "Iron Man 1", as shown below:

```
{
  "title": "Iron Man 1",
  "description": "First movie to kick off the MCU.",
  "rating": "PG-13",
  "shortName": "IM1",
  "sequel": "IM2"
}
```

Another example with a complex result takes in a parameter via `QueryString` and checks for an exact match with a specific property. In this case the `Related()` action method calls the repository's `GetBySequel()` method to find a specific movie by its sequel's short name.

```
// GET: api/ci/related?sequel=IM2
[HttpGet("Related")]
public CinematicItem Related(string sequel)
{
    return _cinematicItemRepository.GetBySequel(sequel);
}
```

The `GetBySequel()` method in the `CinematicItemRepository.cs` class checks for a movie's sequel by the `shortName` parameter and returns the first match.

```

public CinematicItem GetBySequel(string sequelShortName)
{
    return CinematicItems().FirstOrDefault(ci => ci.Sequel ==
sequelShortName);
}

```

While the application is running, navigate to the following endpoint:

- <https://localhost:44372/api/ci/related?sequel=IM3>

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> sequel	IM3	
Key	Value	Description

```

1 [ {
2   "title": "Iron Man 2",
3   "description": "Sequel to the first Iron Man movie.",
4   "rating": "PG-13",
5   "shortName": "IM2",
6   "sequel": "IM3"
7 }]

```

This triggers a GET request by calling the CIController's **Related()** method, with the **sequel** parameter. When passing the sequel's short name "IM3", this returns one item "Iron Man 2", as shown below:

```

{
    "title": "Iron Man 2",
    "description": "Sequel to the first Iron Man movie.",
    "rating": "PG-13",
    "shortName": "IM2",
    "sequel": "IM3"
}

```

As you can see, the result is in JSON format for the returned object.

# XML Serialization

*Wait a minute... with all these JSON results, when will we get to XML serialization?* Not to worry, there are multiple ways to get XML results while reusing the above code. First, add the NuGet package “Microsoft.AspNetCore.Mvc.Formatters.Xml” to your project and then update your Startup.cs file’s **ConfigureServices()** to include a call to services.AddMvc.**AddXmlSerializerFormatters()**:

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

Set the request’s Accept header value to “application/xml” *before* requesting the endpoint, then run the application and navigate to the following endpoint once again:

- <https://localhost:44372/api/ci/IM1>

The screenshot shows the Postman interface with the following details:

- Method:** GET
- URL:** https://localhost:44372/api/ci/IM1
- Headers:** (1)  
Accept: application/xml  
Key: Value
- Body:** (Empty)
- Tests:** (Empty)
- Cookies:** (Empty)
- Comments:** (0)
- Status:** 200 OK | Time: 100 ms | Size: 608 B
- Body (Pretty):**

```
1 <CinematicItem xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2   <Title>Iron Man 1</Title>
3   <Description>First movie to kick off the MCU.</Description>
4   <Rating>PG-13</Rating>
5   <ShortName>IM1</ShortName>
6   <Sequel>IM2</Sequel>
7 </CinematicItem>
```

This should provide the following XML results:

```
<CinematicItem xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Title>Iron Man 1</Title>
  <Description>First movie to kick off the MCU.</Description>
  <Rating>PG-13</Rating>
  <ShortName>IM1</ShortName>
  <Sequel>IM2</Sequel>
</CinematicItem>
```

Since the action method returns a complex object, the result can easily be switched to XML simply by changing the Accept header value. In order to return XML using an IActionResult method, you should also use the **[Produces]** attribute, which can be set to “**application/xml**” at the Controller level

```
[Produces("application/xml")]
[Route("api/[controller]")]
[ApiController]
public class CIController : Controller
{
    ...
}
```

Then revisit the following endpoint, calling the search action method with the fragment parameter set to “ir”:

- <https://localhost:44372/api/ci/search?fragment=ir>

At this point, it is no longer necessary to set the Accept header to “application/xml” during the request, since the **[Produces]** attribute is given priority over it.

The screenshot shows the Postman interface with the following details:

- Method:** GET
- URL:** https://localhost:44372/api/ci/search?fragment=ir
- Params:**

KEY	VALUE	DESCRIPTION
fragment	ir	
Key	Value	Description
- Body:** XML response (Pretty, Raw, Preview) showing the following XML structure:
 

```

1 <ArrayOfCinematicItem xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2   <CinematicItem>
3     <Title>Iron Man 1</Title>
4     <Description>First movie to kick off the MCU.</Description>
5     <Rating>PG-13</Rating>
6     <ShortName>IM1</ShortName>
7     <Sequel>IM2</Sequel>
8   </CinematicItem>
9   <CinematicItem>
10    <Title>Iron Man 2</Title>
11    <Description>Sequel to the first Iron Man movie.</Description>
12    <Rating>PG-13</Rating>
      
```
- Status:** 200 OK
- Time:** 352 ms
- Size:** 1014 B

This should produce the following result, with an array of **CinematicItem** objects in XML:

```

<ArrayOfCinematicItem xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <CinematicItem>
    <Title>Iron Man 1</Title>
    <Description>First movie to kick off the MCU.</Description>
    <Rating>PG-13</Rating>
    <ShortName>IM1</ShortName>
    <Sequel>IM2</Sequel>
  </CinematicItem>
  <CinematicItem>
    <Title>Iron Man 2</Title>
    <Description>Sequel to the first Iron Man movie.</Description>
    <Rating>PG-13</Rating>
    <ShortName>IM2</ShortName>
    <Sequel>IM3</Sequel>
  </CinematicItem>
  <CinematicItem>
    <Title>Iron Man 3</Title>
    <Description>Wraps up the Iron Man trilogy.</Description>
    <Rating>PG-13</Rating>
    <ShortName>IM3</ShortName>
    <Sequel />
  </CinematicItem>
</ArrayOfCinematicItem>
      
```

As for the first **Get()** method returning **JsonResult**, you can't override it with the **[Produces]** attribute or the **Accept** header value to change the result to XML format.

To recap, the order of precedence is as follows:

1. `public JsonResult Get()`
2. `[Produces("application/...")]`
3. `Accept: "application/..."`

## References

- JSON and XML Serialization in ASP.NET Web API – ASP.NET 4.x | Microsoft Docs:  
<https://docs.microsoft.com/en-us/aspnet/web-api/overview/formats-and-model-binding/json-and-xml-serialization#xml-media-type-formatter>
- How to format response data as XML or JSON, based on the request URL in ASP.NET Core:  
<https://andrewlock.net/formatting-response-data-as-xml-or-json-based-on-the-url-in-asp-net-core/>
- Format response data in ASP.NET Core Web API | Microsoft Docs:  
<https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting?view=aspnetcore-2.2>

# YAML-defined CI/CD for ASP .NET Core

By Shahed C on June 24, 2019

2 Replies

This is the **twenty-fifth** of a series of posts on ASP .NET Core in 2019. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**



## In this Article:

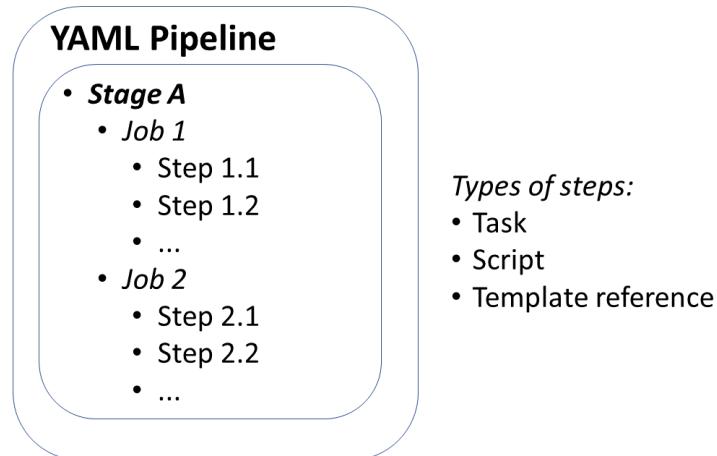
- YAML-defined CI/CD for ASP .NET Core
- Getting Started with Pipelines
- OS/Environment and Runtime
- Restore and Build
- Unit Testing and Code Coverage
- Package and Deploy
- Triggers, Tips & Tricks
- References

# Y is for YAML-defined CI/CD for ASP .NET Core

If you haven't heard of it yet, YAML is yet another markup language. No really, it is. YAML literally stands for Yet Another Markup Language. If you need a reference for YAML syntax and how it applies to Azure DevOps Pipelines, check out the official docs:

- YAML schema – Azure Pipelines: <https://docs.microsoft.com/en-us/azure/devops/pipelines/yaml-schema>

In the context of Azure DevOps, you can use Azure Pipelines with YAML to make it easier for you set up a CI/CD pipeline for Continuous Integration and Continuous Deployment. This includes steps to build and deploy your app. Pipelines consist of stages, which consist of jobs, which consists of steps. Each step could be a script or task. In addition to these options, a step can also be a reference to an external template to make it easier to create your pipelines.



This article will refer to the following sample code on GitHub, which contains a Core 2.2 web project and a sample YAML file:



Web Project with YAML Pipeline : <https://github.com/shahedc/AspNetCoreWithPipeline>

# Getting Started With Pipelines

To get started with Azure Pipelines in Azure DevOps:

1. Log in at: <https://dev.azure.com>
2. Create a **Project** for your **Organization**
3. Add a new Build Pipeline under **Pipelines | Builds**
4. **Connect** to your code location, e.g. GitHub repo
5. **Select** your repo, e.g. a specific GitHub repository
6. **Configure** your YAML
7. **Review** your YAML and **Run** it

From here on forward, you may come back to your YAML here, edit it, save it, and run as necessary. You'll even have the option to commit your YAML file "azure-pipelines.yml" into your repo, either in the master branch or in a separate branch (to be submitted as a Pull Request that can be merged).

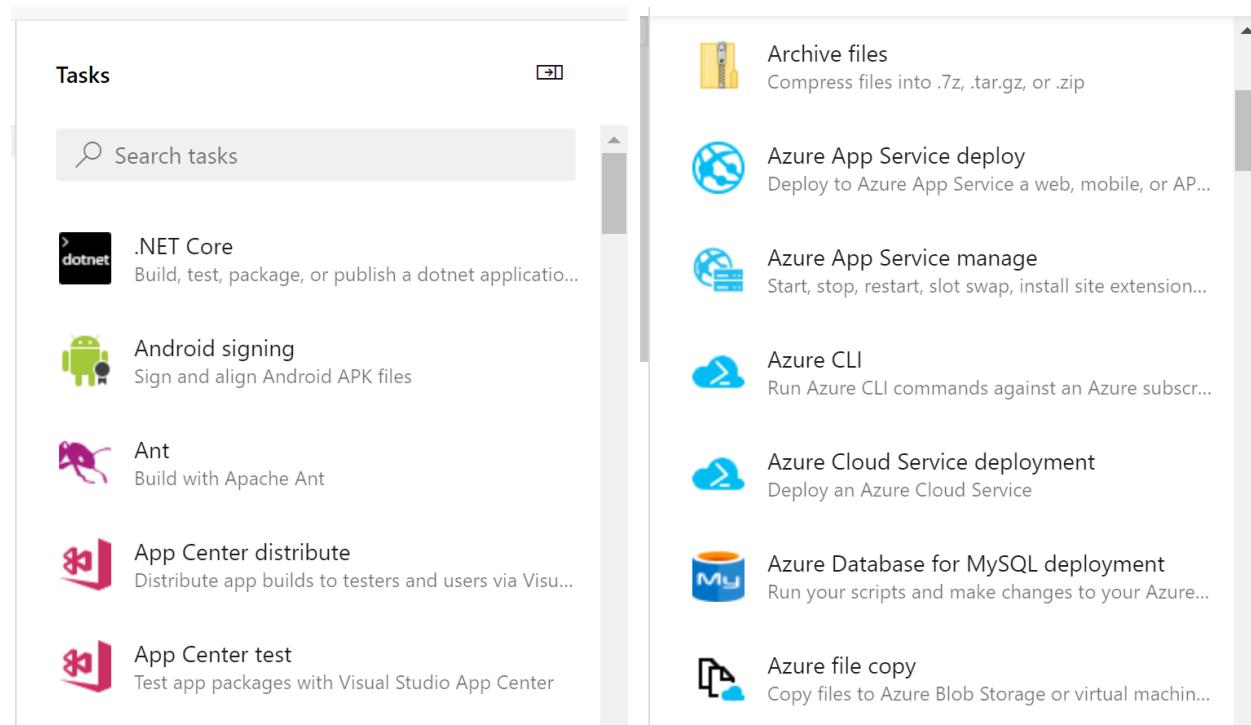
The screenshot shows the Azure DevOps interface for creating a new pipeline. The left sidebar has 'ShahedWebProject' selected under 'Pipelines'. The top navigation bar shows 'ShahedWebProject / Pipelines'. The main area has tabs: 'Connect' (checked), 'Select' (checked), 'Configure' (checked), and 'Review' (underlined). A sub-header says 'New pipeline'. Below it, the title 'Review your pipeline YAML' is displayed. A code editor shows the 'azure-pipelines.yml' file with the following content:

```
6 trigger:
7 - master
8
9 pool:
10   vmImage: 'windows-2019'
11
12 variables:
13   buildConfiguration: 'Release'
14
15 steps:
16   - script: dotnet restore
17
```

If you need more help getting started, check out the official docs and Build 2019 content at:

- Create your first pipeline: <https://docs.microsoft.com/en-us/azure/devops/pipelines/create-first-pipeline>
- Build, test, and deploy .NET Core apps: <https://docs.microsoft.com/en-us/azure/devops/pipelines/languages/dotnet-core>
- From Build 2019, “YAML Release Pipelines in Azure DevOps”: <https://youtu.be/ORy3OeqLZIE>

To add pre-written snippets to your YAML, you may use the Task Assistant side panel to insert a snippet directly into your YAML file. This includes tasks for .NET Core builds, Azure App Service deployment and more.



## OS/Environment and Runtime

From the sample repo, take a look at the sample YAML file “azure-pipelines.yml”. Near the top, there is a definition for a “**pool**” with a “**vmImage**” set to ‘windows-2019’.

```
pool:  
  vmImage: 'windows-2019'
```

If I had started off with the default YAML pipeline configuration for a .NET Core project, I would probably get a **vmImage** value set to ‘ubuntu-latest’. This is just one of many possible values. From the official docs on Microsoft-hosted agents, we can see that Microsoft’s agent pool provides at least the following VM images across multiple platforms, e.g.

- Visual Studio 2019 Preview on Windows Server 2019 (windows-2019)
- Visual Studio 2017 on Windows Server 2016 (vs2017-win2016)
- Visual Studio 2015 on Windows Server 2012R2 (vs2015-win2012r2)
- Windows Server 1803 (win1803) – for running Windows containers
- macOS X Mojave 10.14 (macOS-10.14)
- macOS X High Sierra 10.13 (macOS-10.13)
- Ubuntu 16.04 (ubuntu-16.04)

In addition to the OS/Environment, you can also set the .NET Core runtime version. This may come in handy if you need to explicitly set the runtime for your project.

```
steps:  
- task: DotNetCoreInstaller@0  
  inputs:  
    version: '2.2.0'
```

## Restore and Build

Once you’ve set up your OS/environment and runtime, you can restore and build your project. To build a specific configuration by name, you can set up a variable first to define the build configuration, and then pass in the variable name to the build step.

```
variables:  
  buildConfiguration: 'Release'
```

```
steps:
- script: dotnet restore

- script: dotnet build --configuration $(buildConfiguration)
  displayName: 'dotnet build $(buildConfiguration)'
```

In the above snippet, the **buildConfiguration** is set to ‘Release’ so that the project is built for its ‘Release’ configuration. The **displayName** is a friendly name in a text string (for any step) that may include variable names as well. This is useful for observing logs and messages during troubleshooting and inspection.

Note the use of **script** steps to make use of **dotnet** commands with parameters you may already be familiar with, if you’ve been using .NET Core CLI Commands. This makes it easier to run steps without having to spell everything out. From the official docs, here are some more detailed steps for restore and build, if you wish to customize your steps and tasks further:

```
steps:
- task: DotNetCoreCLI@2
  inputs:
    command: restore
    projects: '**/*.csproj'
    feedsToUse: config
    nugetConfigPath: NuGet.config
    externalFeedCredentials: <Name of the NuGet service connection>
```

Note that you can set your own values for an external NuGet feed to restore dependencies for your project. Once restored, you may also customize your build steps/tasks.

```
steps:
- task: DotNetCoreCLI@2
  displayName: Build
  inputs:
    command: build
    projects: '**/*.csproj'
    arguments: '--configuration Release'
```

## Unit Testing and Code Coverage

Although unit testing is not required for a project to be compiled and deployed, it is absolutely essential for any real-world application. In addition to running unit tests, you may also want to measure your code coverage for those unit tests. All these are possible via YAML configuration.

From the official docs, here is a snippet to run your unit tests, that is equivalent to a “**dotnet test**” command for your project:

```
steps:  
- task: DotNetCoreCLI@2  
  inputs:  
    command: test  
    projects: '**/*Tests/*.csproj'  
    arguments: '--configuration $(buildConfiguration)'
```

Also, here is another snippet to collect code coverage:

```
steps:  
- task: DotNetCoreCLI@2  
  inputs:  
    command: test  
    projects: '**/*Tests/*.csproj'  
    arguments: '--configuration $(buildConfiguration) --collect "Code coverage"'
```

Once again, the above snippet uses the “**dotnet test**” command, but also adds the **-collect** option to enable the data collector for your test run. The text string value that follows is a friendly name that you can set for the data collector. For more information on “**dotnet test**” and its options, check out the docs at:

- dotnet test command – .NET Core CLI: <https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-test#options>

## Package and Deploy

Finally, it’s time to package and deploy your application. In this example, I am deploying my web app to Azure App Service.

```
- task: DotNetCoreCLI@2  
  inputs:  
    command: publish  
    publishWebProjects: True  
    arguments: '--configuration $(BuildConfiguration) --output  
$(Build.ArtifactStagingDirectory)'  
    zipAfterPublish: True
```

```
- task: PublishBuildArtifacts@1
  displayName: 'publish artifacts'
```

The above snippet runs a “**dotnet publish**” command with the proper configuration setting, followed by an output location, e.g. `Build.ArtifactStagingDirectory`. The value for the output location is one of many predefined build/system variables, e.g. `System.DefaultWorkingDirectory`, `Build.StagingDirectory`, `Build.ArtifactStagingDirectory`, etc. You can find out more about these variables from the official docs:

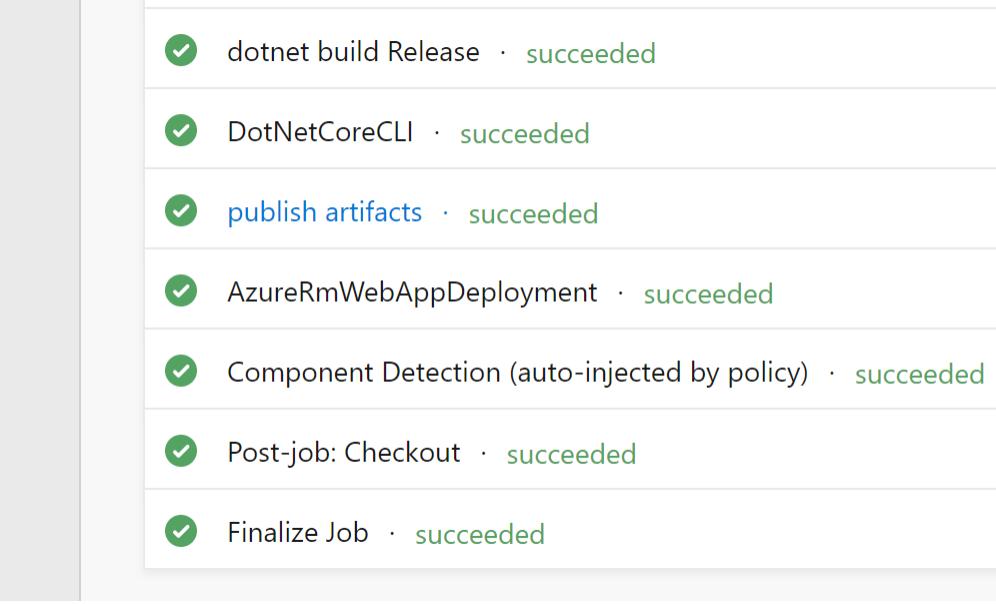
- Predefined variables – Azure Pipelines: <https://docs.microsoft.com/en-us/azure/devops/pipelines/build/variables>

The **PublishBuildArtifacts** task uploads the package to a file container, ready for deployment. After your artifacts are ready, it’s time to deploy your web app to Azure, e.g. Azure App Service.

```
- task: AzureRmWebAppDeployment@4
  inputs:
    ConnectionType: 'AzureRM'
    azureSubscription: '<REPLACE_AZURE_SUBSCRIPTION_NAME_(ID)>'
    appType: 'webApp'
    WebAppName: 'WebProjectForPipelines'
    packageForLinux: '$(System.ArtifactsDirectory)/**/*.zip'
    enableCustomDeployment: true
    DeploymentType: 'webDeploy'
```

The above snippet runs **msdeploy.exe** using the previously-created zipped package. Note that there is a placeholder text for the Azure Subscription ID. If you use the Task Assistant panel to add a “Azure App Service Deploy” snippet, you will be prompted to select your Azure Subscription, and a Web App location to deploy to, including deployment slots if necessary. Note that the **DeploymentType** actually defaults to ‘webDeploy’ so setting the value may not be necessary. However, if **UseWebDeploy (optional)** is set to true, the **DeploymentType** is required.

You may use the Azure DevOps portal to inspect the progress of each step and troubleshoot any failed steps. You can also drill down into each step to see the commands that are running in the background, followed by any console messages.

- 
- The screenshot shows a list of pipeline steps with their outcomes:
- dotnet build Release · succeeded
  - DotNetCoreCLI · succeeded
  - publish artifacts · succeeded
  - AzureRmWebAppDeployment · succeeded
  - Component Detection (auto-injected by policy) · succeeded
  - Post-job: Checkout · succeeded
  - Finalize Job · succeeded

**NOTE:** to set up a release pipeline with multiple stages and optional approval conditions, check out the official docs at:

- (Classic) Release pipelines: <https://docs.microsoft.com/en-us/azure/devops/pipelines/release>
- (2019) Announcement: <https://devblogs.microsoft.com/devops/whats-new-with-azure-pipelines/>

## Triggers, Tips & Tricks

Now that you've set up your pipeline, how does this all get triggered? If you've taken a look at the sample YAML file, you will notice that the first command includes a **trigger**, followed by the word "master". This ensures that the pipeline will be triggered every time code is pushed to the corresponding code repository's *master* branch. When using a template upon creating the YAML file, this trigger should be automatically included for you.

```
trigger:  
- master
```

To include more triggers, you may specify triggers for specific branches to include or exclude.

```
trigger:  
branches:  
include:  
- master  
- releases/*  
exclude:  
- releases/old*
```

Finally here are some tips and tricks when using YAML to set up CI/CD using Azure Pipelines:

- **Snippets:** when you use the Task Assistant panel to add snippets into your YAML, be careful where you are adding each snippet. It will insert it wherever your cursor is positioned, so make sure you've clicked into the correct location before inserting anything.
- **Order of tasks and steps:** Verify that you've inserted (or typed) your tasks and steps in the correct order. For example: if you try to deploy an app before publishing it, you will get an error.
- **Indentation:** Whether you're typing your YAML or using the snippets (or some other tool), use proper indentation. You will get syntax errors if the steps and tasks aren't indented correctly.
- **Proper Runtime/OS:** Assign the proper values for the desired runtime, environment and operating system.
- **Publish Artifacts:** Don't forget to publish your artifacts before attempting to deploy the build.
- **Artifacts location:** Specify the proper location(s) for artifacts when needed.
- **Authorize Permissions:** When connecting your Azure Pipeline to your code repository (e.g. GitHub repo) and deployment location (e.g. Azure App Service), you will be prompted to authorize the appropriate permissions. Be aware of what permissions you're granting.
- **Private vs Public:** Both your Project and your Repo can be private or public. If you try to mix and match a public Project with a private Repo, you will get the following warning message: "*You selected a private repository, but this is a public project. Go to project settings to change the visibility of the project.*"

## References

- YAML schema: <https://docs.microsoft.com/en-us/azure/devops/pipelines/yaml-schema>
- Deploy an Azure Web App: <https://docs.microsoft.com/en-us/azure/devops/pipelines/targets/webapp>

- [VIDEO] YAML Release Pipelines in Azure DevOps:  
<https://www.youtube.com/watch?v=ORy3OeqLZIE>
- Microsoft-hosted agents for Azure Pipelines: <https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/hosted>
- Build, test, and deploy .NET Core apps: <https://docs.microsoft.com/en-us/azure/devops/pipelines/languages/dotnet-core>
- Create your first pipeline: <https://docs.microsoft.com/en-us/azure/devops/pipelines/create-first-pipeline>
- Getting Started with YAML: <https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started-yaml>
- Job and step templates: <https://docs.microsoft.com/en-us/azure/devops/pipelines/process/templates>
- Build pipeline triggers: <https://docs.microsoft.com/en-us/azure/devops/pipelines/build/triggers>
- Release variables and debugging: <https://docs.microsoft.com/en-us/azure/devops/pipelines/release/variables>
- Make your project public or private: <https://docs.microsoft.com/en-us/azure/devops/organizations/public/make-project-public>
- Azure App Service Deploy task: <https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/deploy/azure-rm-web-app-deployment>
- Publish Build Artifacts task: <https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/utility/publish-build-artifacts>

# Zero-Downtime\* Web Apps for ASP .NET Core

By Shahed C on July 1, 2019

5 Replies

This is the **twenty-sixth** of a series of posts on ASP .NET Core in 2019. In this series, we've cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**

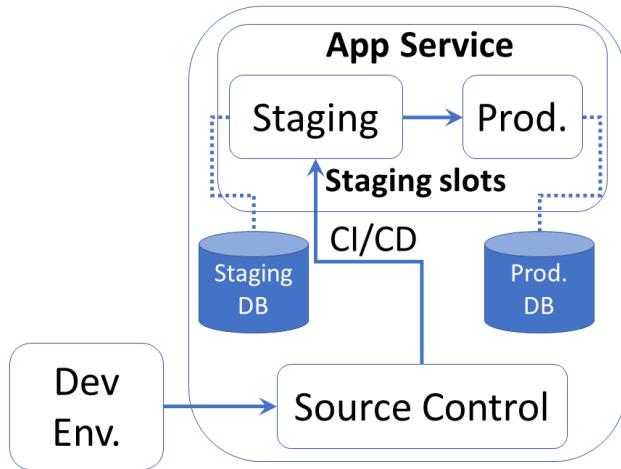


## In this Article:

- Z is for Zero-Downtime\* Web Apps
- Availability
- Backup & Restore
- Continuous Integration & Continuous Deployment
- Deployment Slots
- EF Core Migrations in Production
- Feature Flags
- References

# Z is for Zero-Downtime\* Web Apps for ASP .NET Core

If you've made it this far in this ASP .NET Core A-Z series, hopefully you've learned about many important topics related to ASP .NET Core web application development. As we wrap up this series with a look at tips and tricks to *attempt* zero-downtime, this last post itself has its own lettered A-F mini-series: **A**vailability, **B**ackup & Restore, **C**I/CD, **D**eployment Slots, **E**F Core Migrations and **F**eature Flags.



\* While it may not be possible to get 100% availability 24/7/365, you can ensure a user-friendly experience free from (or at least with minimal) interruptions, by following a combination of the tips and tricks outlined below. This write-up is not meant to be a comprehensive guide. Rather, it is more of an outline with references that you can follow up on, for next steps.

## Availability

To improve the availability of your ASP .NET Core web app running on Azure, consider running your app in multiple regions for HA (High Availability). To control traffic to/from your website, you may use Traffic Manager to direct web traffic to a standby/secondary region, in case the primary region is unavailable.

Consider the following 3 options, in which the primary region is always active and the secondary region may be passive (as a hot or cold standby) or active. When both are active, web requests are load-balanced between the two regions.

Options	Primary Region	Secondary Region
---------	----------------	------------------

A	Active	Passive, Hot Standby
B	Active	Passive, Cold Standby
C	Active	Active

If you're running your web app in a Virtual Machine (VM) instead of Azure App Service, you may also consider Availability Sets. This helps build redundancy in your Web App's architecture, when you have 2 or more VMs in an Availability Set. For added resiliency, use Azure Load Balancer with your VMs to load-balance incoming traffic. As an alternative to Availability Sets, you may also use Availability Zones to counter any failures within a datacenter.

## Backup & Restore

Azure's App Service lets you back up and restore your web application, using the Azure Portal or with Azure CLI commands. Note that this requires your App Service to be in at least the Standard or Premium tier, as it is not available in the Free/Shared tiers. You can create backups on demand when you wish, or schedule your backups as needed. If your site goes down, you can quickly restore your last good backup to minimize downtime.

The screenshot shows the Azure portal interface for managing backups of an App Service application named 'scwebapp2019'. The left sidebar has a search bar and links for Deployment (Quickstart, Deployment slots, Deployment Center), Settings (Configuration, Authentication / Authorization, Application Insights, Identity), and Backups. The 'Backups' link is currently selected. The main content area displays the 'Snapshot (Preview)' section, which informs users that snapshots automatically create periodic restore points. It includes a 'Restore' button. Below it is the 'Backup' section, which allows users to configure backup settings to create restorable archive copies of their app's content, configuration, and database. A message indicates that a backup was configured and started on Monday, July 1, 2019, at 12:19:50 AM EDT, with a backup occurring every 1 Day(s). It features 'Backup' and 'Restore' buttons. A note at the bottom states 'No backup history is available'.

In addition to the app itself, the backup process also backs up the Web App's configuration, file contents and the database connected to your app. Database types include SQL DB (aka SQL Server PaaS), MySQL and PostgreSQL. Note that these backups include a *complete* backup, and not incremental/delta backups.

## Continuous Integration & Continuous Deployment

In the previous post, we covered CI/CD with YAML pipelines. Whether you have to fix an urgent bug quickly or just deploy a planned release, it's important to have a proper CI/CD pipeline. This allows you to deploy new features and fixes quickly with minimal downtime.

The screenshot shows the Azure DevOps interface for a project named "ShahedWebProject". The left sidebar has "Pipelines" selected. The top navigation bar shows "ShahedWebProject / Pipelines". The main area is titled "Review your pipeline YAML" and displays the following YAML code:

```
azure-pipelines.yml
6 trigger:
7 - master
8
9 pool:
10   vmImage: 'windows-2019'
11
12 variables:
13   buildConfiguration: 'Release'
14
15 steps:
16 - script: dotnet restore
17
```

- YAML-defined CI/CD for ASP .NET Core: [https://wakeupandcode.com/yaml-defined-cicd-for-asp-.net-core/](https://wakeupandcode.com/yaml-defined-cicd-for-asp-net-core/)

## Deployment Slots

Whether you're deploying your Web App to App Service for the first time or the 100th time, it helps to test out your app before releasing to the public. Deployment slots make it easy to set up a *Staging Slot*, warm it up and swap it immediately with a *Production Slot*. Swapping a slot that has already been warmed up ahead of time will allow you to deploy the latest version of your Web App almost immediately.

The screenshot shows the Azure portal interface for an App Service named 'sc2019webapp'. On the left, a sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Deployment, Quickstart, Deployment slots (which is selected and highlighted in blue), and Deployment Center. The main content area is titled 'Deployment Slots' and contains a brief description: 'Deployment slots are live apps with their own hostnames. App content and configurations elements can be swapped between two deployment slots, including the production slot.' Below this is a table with four columns: NAME, STATUS, APP SERVICE PLAN, and TRAFFIC %. The table shows two rows: one for 'sc2019webapp' (Status: Running, App Service Plan: NonFreeAsp, Traffic: 100%) and one for 'sc2019webapp-StagingSlot' (Status: Running, App Service Plan: NonFreeAsp, Traffic: 0%). At the top of the main area are buttons for Save, Discard, Add Slot, Swap, and Refresh.

NAME	STATUS	APP SERVICE PLAN	TRAFFIC %
sc2019webapp	PRODUCTION	Running	NonFreeAsp 100
sc2019webapp-StagingSlot	Running	NonFreeAsp 0	

Note that this feature is only available in Standard, Premium or Isolated App Service tiers, as it is *not* available in the Free/Shared tiers. You can combine Deployment Slots with your CI/CD pipelines to ensure that your automated deployments end up in the intended slots.

## EF Core Migrations in Production

We covered EF Core Migrations in a previous post, which is one way of upgrading your database in various environments (including production). *But wait, is it safe to run EF Core Migrations in a production environment?* Even though you can use auto-generated EF Core migrations (written in C# or outputted as SQL Scripts), you may also modify your migrations for your needs.

I would highly recommend reading Jon P Smith's two-part series on "Handling Entity Framework Core database migrations in production":

- Part 1 of 2: <https://www.thereformedprogrammer.net/handling-entity-framework-core-database-migrations-in-production-part-1/>
- Part 2 of 2: <https://www.thereformedprogrammer.net/handling-entity-framework-core-database-migrations-in-production-part-2/>

What you decide to do is up to you (and your team). I would suggest exploring the different options available to you, to ensure that you minimize any downtime for your users. For any *non-breaking* DB changes, you should be able to migrate your DB easily. However, your site may be down for maintenance for any *breaking* DB changes.

## Feature Flags

Introduced by the Azure team, the Microsoft.FeatureManagement package allows you to add Feature Flags to your .NET application. This enables your web app to include new features that can easily be toggled for various audiences. This means that you could potentially test out new features by deploying them during off-peak times, but toggling them to become available via app configuration.

To install the package, you may use the following **dotnet** command:

```
>dotnet add package Microsoft.FeatureManagement --version 1.0.0-  
preview-XYZ
```

... where XYZ represents the a specific version number suffix for the latest preview. If you prefer the Package Manager Console in Visual Studio, you may also use the following PowerShell command:

```
>Install-Package Microsoft.FeatureManagement -Version 1.0.0-preview-  
XYZ
```

By combining many/all of the above features, tips and tricks for your Web App deployments, you can release new features while minimizing/eliminating downtime. If you have any new suggestions, feel free to leave your comments.

## References

- Highly available multi-region web application: <https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/app-service-web-app/multi-region>
- Design reliable Azure applications: <https://docs.microsoft.com/en-us/azure/architecture/reliability/>

- Manage the availability of Windows VMs in Azure: <https://docs.microsoft.com/en-us/azure/virtual-machines/windows/manage-availability>
- What is Azure Load Balancer? <https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview>
- SLA for VMs: <https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/>
- Back up app – Azure App Service: <https://docs.microsoft.com/en-us/azure/app-service/manage-backup>
- Azure CLI Script Sample – Back up an app: <https://docs.microsoft.com/en-us/azure/app-service/scripts/cli-backup-onetime>
- CI/CD with Release pipelines: <https://docs.microsoft.com/en-us/azure/devops/pipelines/release>
- Continuous deployment – Azure App Service: <https://docs.microsoft.com/en-us/azure/app-service/deploy-continuous-deployment>
- Set up staging environments for web apps in Azure App Service: <https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots>
- Handling Entity Framework Core database migrations in production: <https://www.thereformedprogrammer.net/handling-entity-framework-core-database-migrations-in-production-part-1/>
- Handling Entity Framework Core database migrations in production – Part 2: <https://www.thereformedprogrammer.net/handling-entity-framework-core-database-migrations-in-production-part-2/>
- Tutorial for using feature flags in a .NET Core app: <https://docs.microsoft.com/en-us/azure/azure-app-configuration/use-feature-flags-dotnet-core>
- Quickstart for adding feature flags to ASP.NET Core: <https://docs.microsoft.com/en-us/azure/azure-app-configuration/quickstart-feature-flag-aspnet-core>