As seen on
WakeUpAndCode.com

26 topics for ASP.NET Core Web App Development

3.1!

ASP.NET Core A-Z

Shahed Chowdhuri

Authentication & Authorization in ASP .NET Core 3.1

By Shahed C on January 6, 2020

13 Replies

This is the first of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

If you need some guidance before you get started with this series, check out my late 2019 posts, which serve as a prelude to the 2020 series:

- ASP .NET Core code sharing between Blazor, MVC and Razor Pages
- Hello ASP .NET Core v3.1!
- NetLearner on ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.1-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.1-alpha

In this Article:

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 one of the pre-built templates with one of the Authentication options. The examples below demonstrate both the CLI commands and Visual Studio UI.

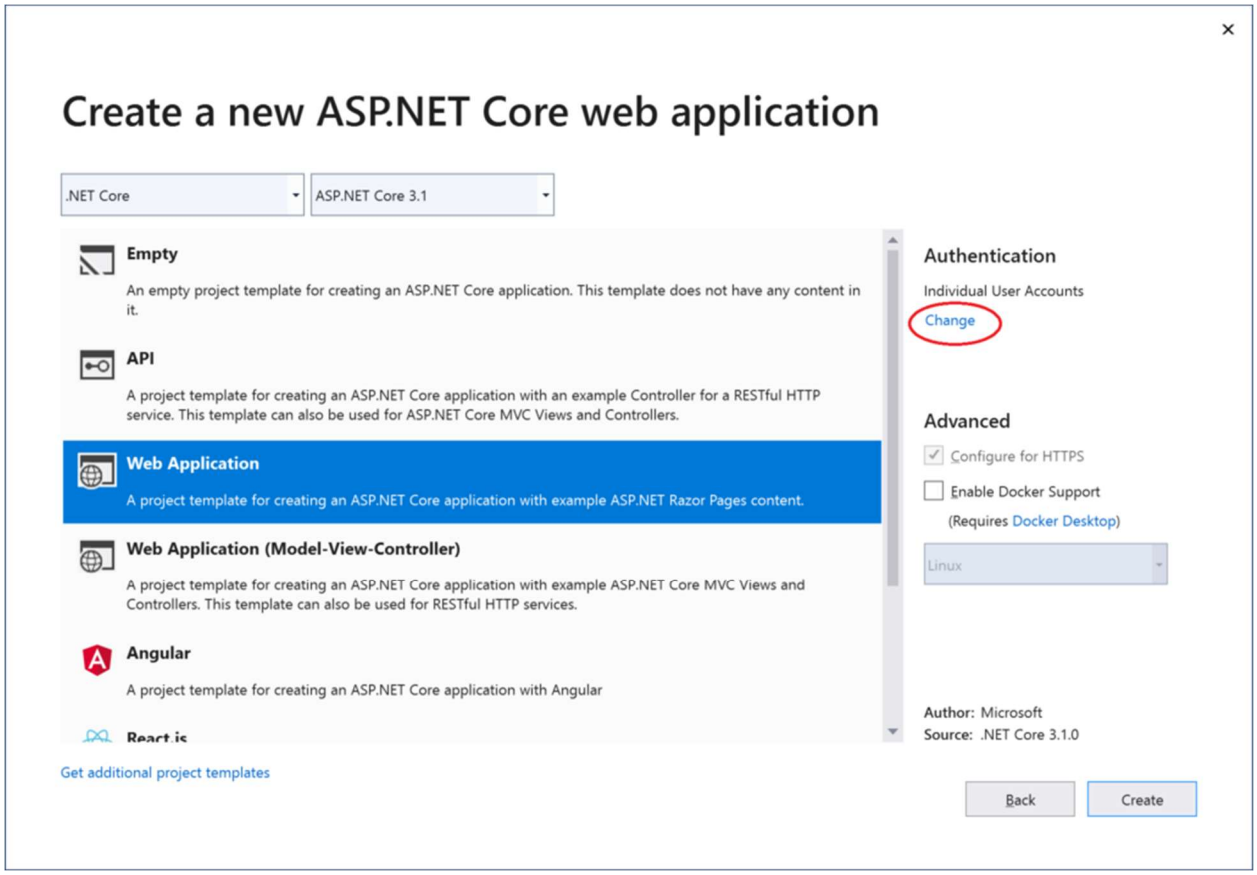
Here are the CLI Commands for MVC, Razor Pages and Blazor (Server), respectively:

```
> dotnet new mvc --auth Individual -o mvcsample
> dotnet new webapp --auth Individual -o pagessample
> dotnet new blazorserver --auth Individual -o blazorsample
```

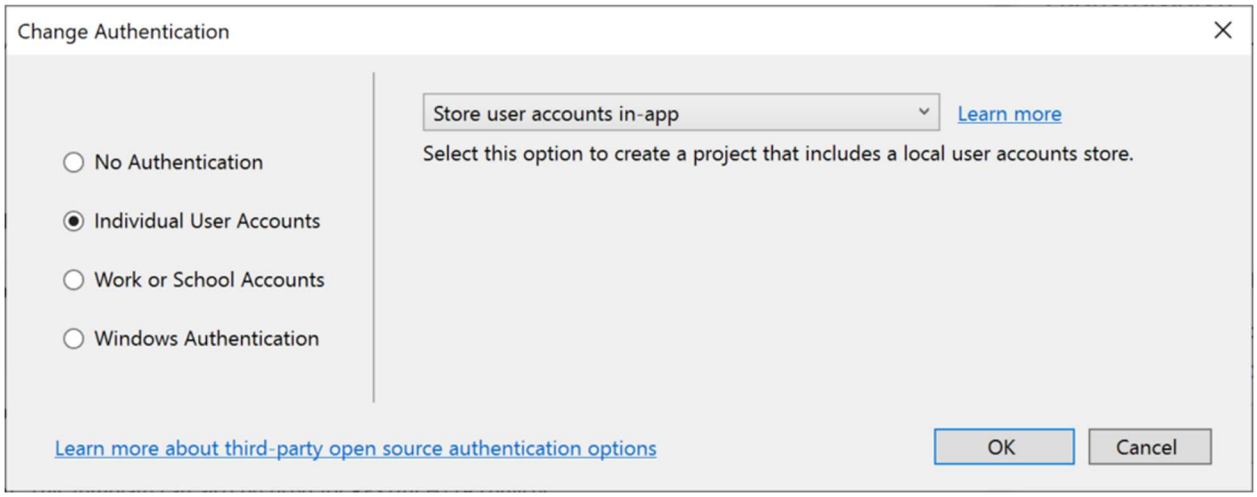
Things to note:

- The **dotnet new** command is followed by the template name (mvc, webapp, blazorserver).
- The –auth option allows you to specify the authentication type, e.g. Individual
- The -o option is an optional parameter that provides the output folder name for the project to be created in.

Here is the opening dialog in Visual Studio 2019, for creating a 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 2019 post on Razor UI Libraries, and jump to the last bonus 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 MVC app, the following snippets of code are added to the Startup.cs configuration:

```
public void ConfigureServices(IServiceCollection services)
{
...
   services.AddDbContext<LibDbContext>(options =>
   {
      options

.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"),
         assembly =>
         assembly.MigrationsAssembly
            (typeof(LibDbContext).Assembly.FullName));
   });

   services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
   .AddEntityFrameworkStores<LibDbContext>();
...
} public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
{
...
 app.UseStaticFiles();
...
 app.UseAuthentication();
 app.UseAuthorization();
...
 app.Endpoints(...);
}
```

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 calls to app.UseAuthentication and app.UseAuthorization to ensure that authentication and authorization are used by your web app. Note that this appears *after* app.UseStaticFiles() but *before* app.UseEndpoints() 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.
- Similar to the MVC web project, you can also browse the Startup.cs file for the equivalent Razor Pages and Blazor web app projects.

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.AddRazorPages()
     .AddRazorPagesOptions(options =>
     {

options.Conventions.AuthorizePage("/LearningResources/Create");
         options.Conventions.AuthorizePage("/LearningResources/Edit");

options.Conventions.AuthorizePage("/LearningResources/Delete");
         options.Conventions.AuthorizePage("/ResourceLists/Create");
         options.Conventions.AuthorizePage("/ResourceLists/Edit");
         options.Conventions.AuthorizePage("/ResourceLists/Delete");
         options.Conventions.AllowAnonymousToPage("/Index");
     });
```

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
{
   ...
}
```

Authorization in Blazor

In a Blazor project, you can wrap elements in an <AuthorizeView> component, which may contain nested elements named <Authorized>, <NotAuthorized> and <Authorizing>.

```
<AuthorizeView>
   <Authorized Context="Auth">
     authorized elements go here
   </Authorized>
   <NotAuthorized>
     anonymous-accessed elements
   </NotAuthorized>
   <Authorizing>
     waiting message...
   </Authorizing>
</AuthorizeView>
```

The root element here has the following characteristics:

- <**AuthorizeView**> element used as a container for nested elements
- Optional role attribute, e.g. <**AuthorizeView** Roles="Admin,RoleA"> used to set a comma-separated list roles that will be used to determine who can view the authorized nested elements
- Optional context attribute, e.g. <**AuthorizeView** Context="Auth"> to define a custom context, to avoid any conflicts with nested comments

The nested elements represent the following:

- **Authorized**: shown to users who have been authorized to view and interact with these elements
- **NotAuthorized**: shown to users who have *not* been authorized
- **Authorizing**: temporary message shown while authorization is being processed

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 2018 post on EF Core Migrations to learn how you can do the same in your environment.

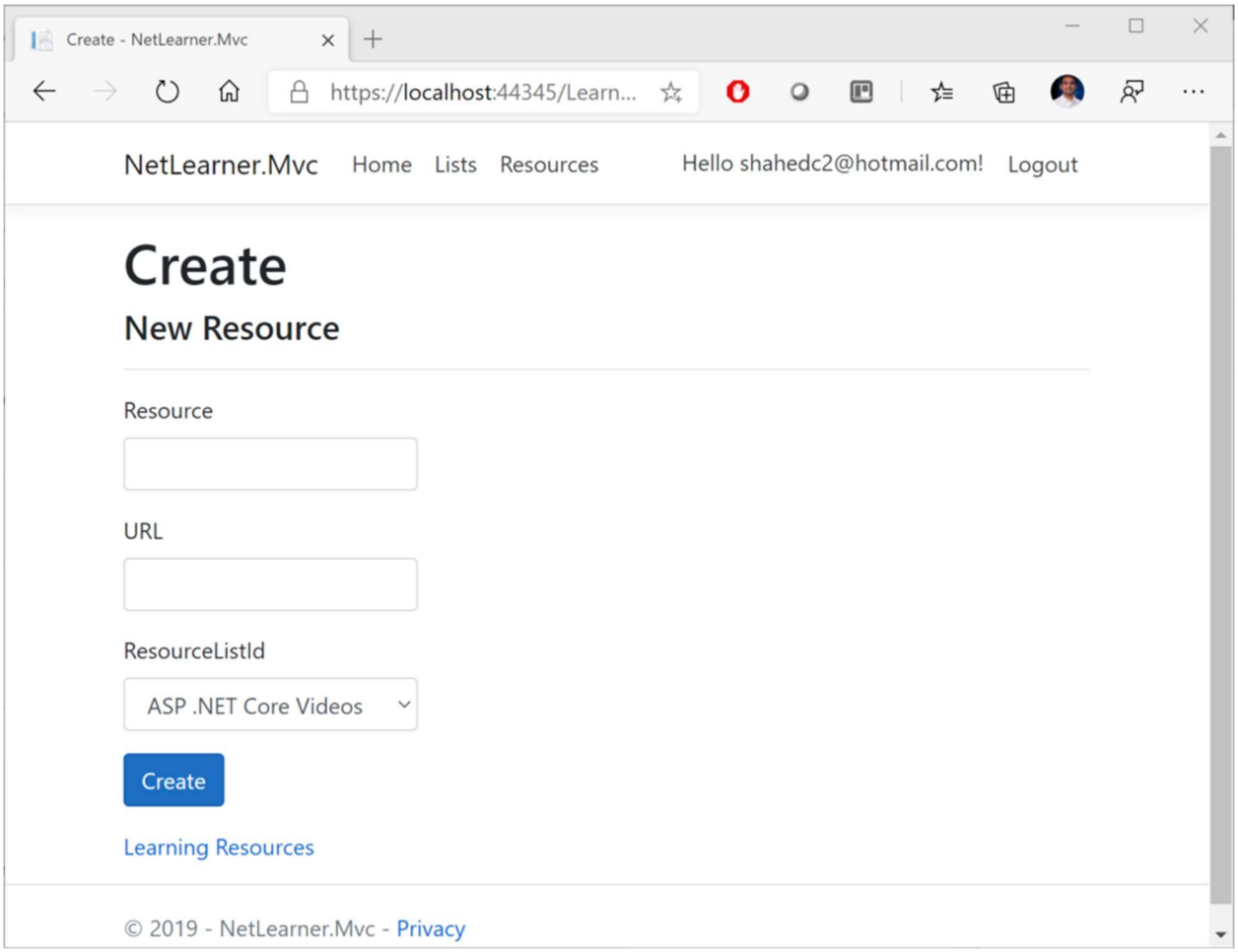
NOTE: the registration feature for each web app has been disabled by default. To enable registration, please do the following:

1. Locate scaffolded Identity pages under /Areas/Identity/Pages/Account/

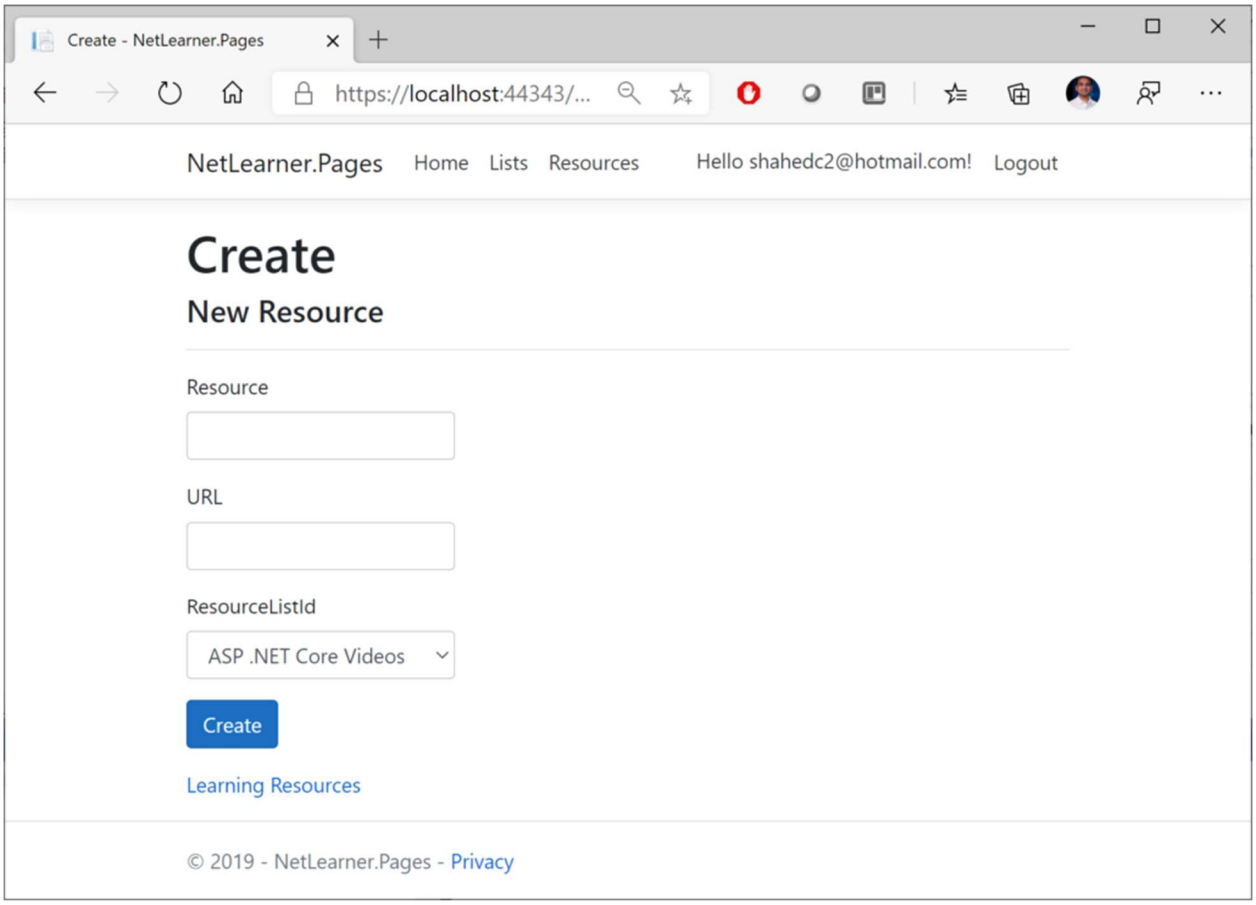
- MVC
- Razor Pages
- Blazor

2. In Register.cshtml, update the page to include environments in addition to Development, if desired.
3. In Register.cshtml.cs, replace [**Authorize**] with [**AllowAnonymous**] to allow access to registration

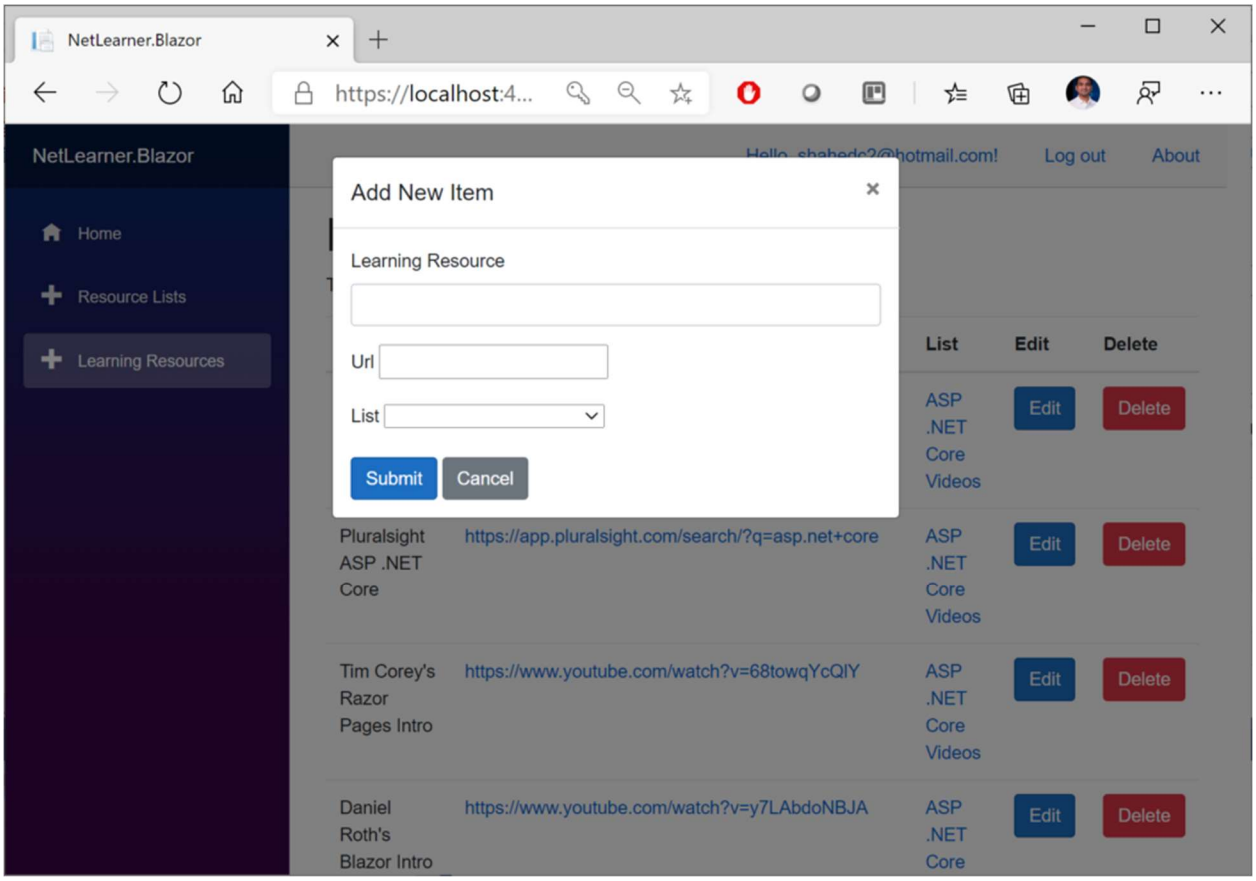
Here are the screenshots of the Create page for a user who is logged in:



NetLearner MVC: Create New Resource

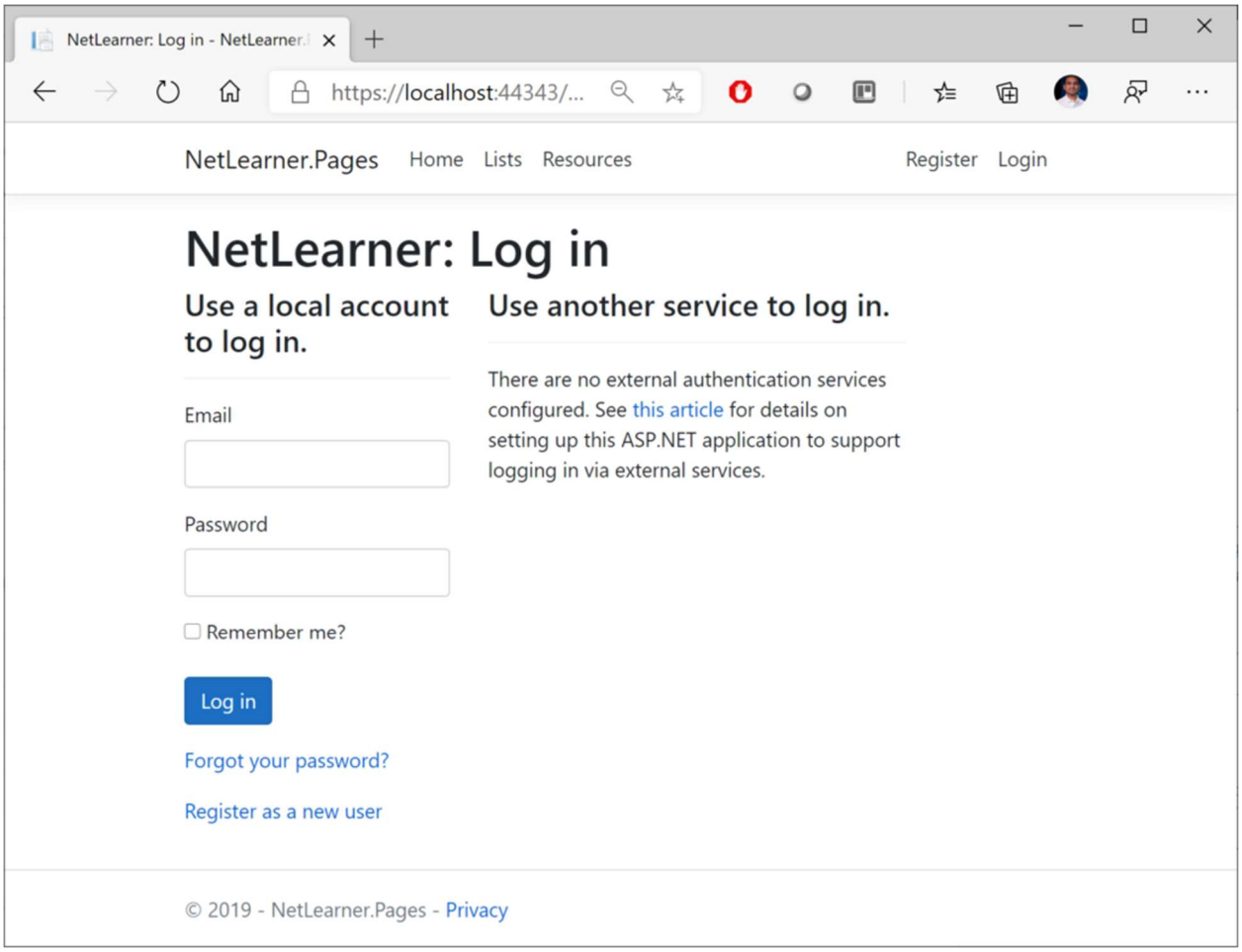


NetLearner Razor Pages: Create New Resource



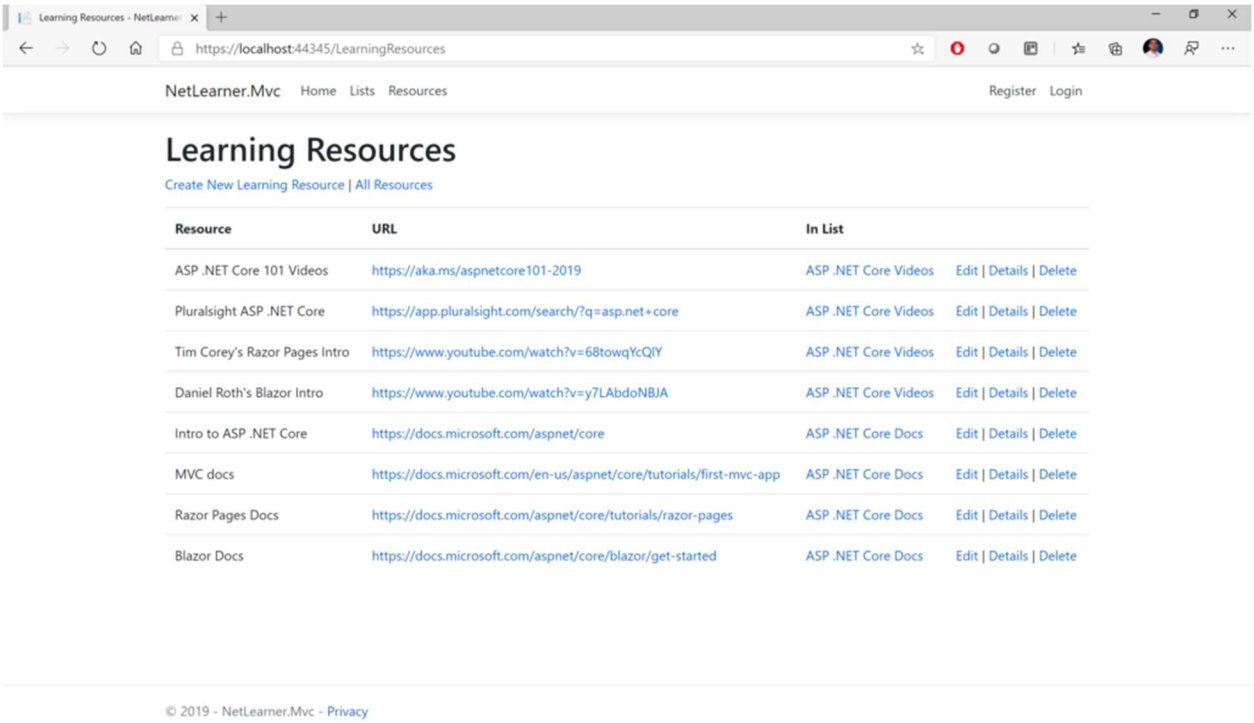
NetLearner Blazor: Create New Resource

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. Note that all 3 web apps (MVC, Razor Pages and Blazor) have similar Identity pages. To allow manual customization, they were also auto-generated via scaffolding and included in all 3 projects.

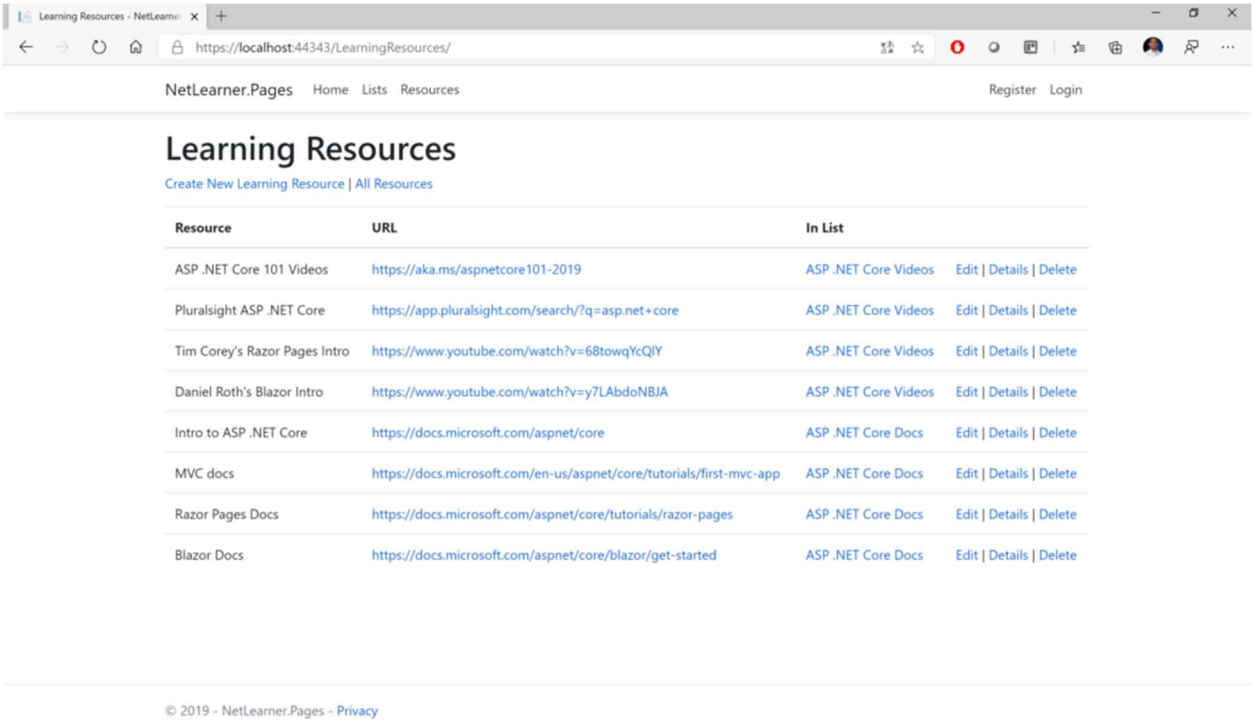


NetLearner: Log in

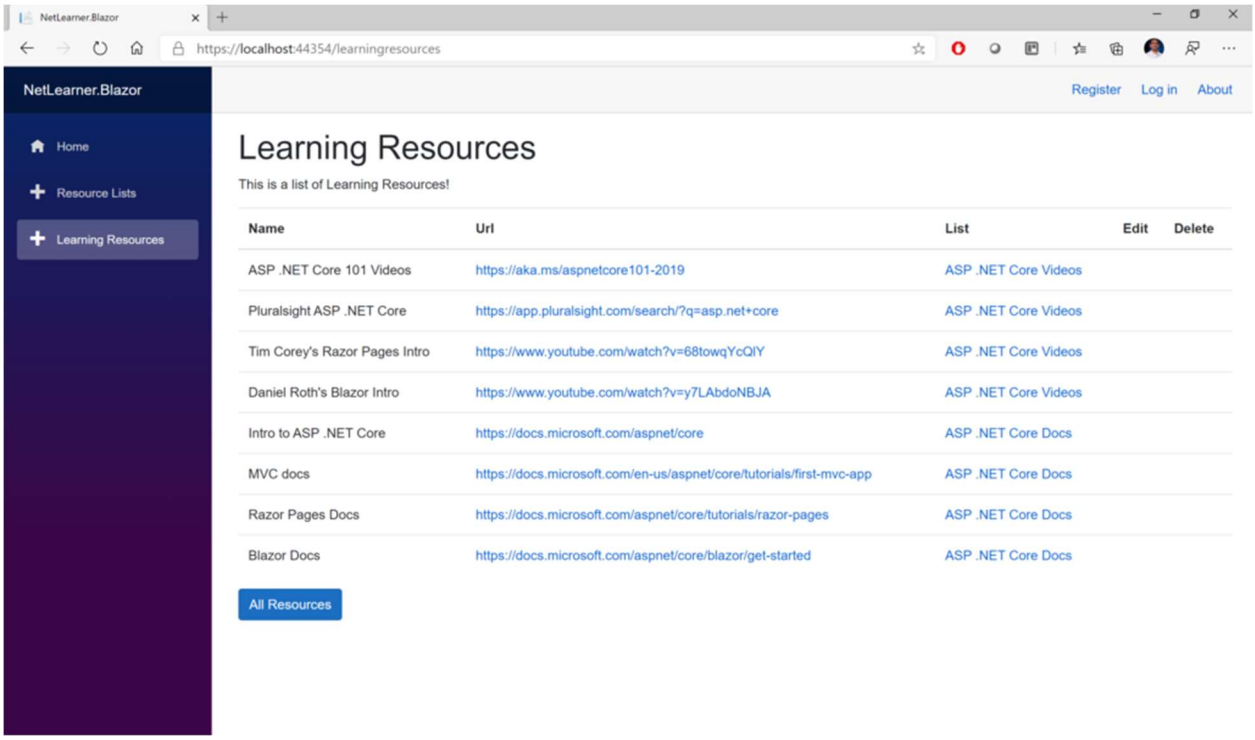
Here are the screenshots showing the list of Learning Resources, visible to anyone whether they're logged in or not:



NetLearner MVC: List of Learning Resources



NetLearner Razor Pages: List of Learning Resources



NetLearner Blazor: List of Learning Resources

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/core/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 Authentication & Authorization: https://docs.microsoft.com/en-us/aspnet/core/security/blazor/?view=aspnetcore-3.1

Blazor Full-Stack Web Dev in ASP .NET Core 3.1

By Shahed C on January 13, 2020

11 Replies

This is the second of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Authentication & Authorization in ASP .NET Core 3.1

NetLearner on GitHub:

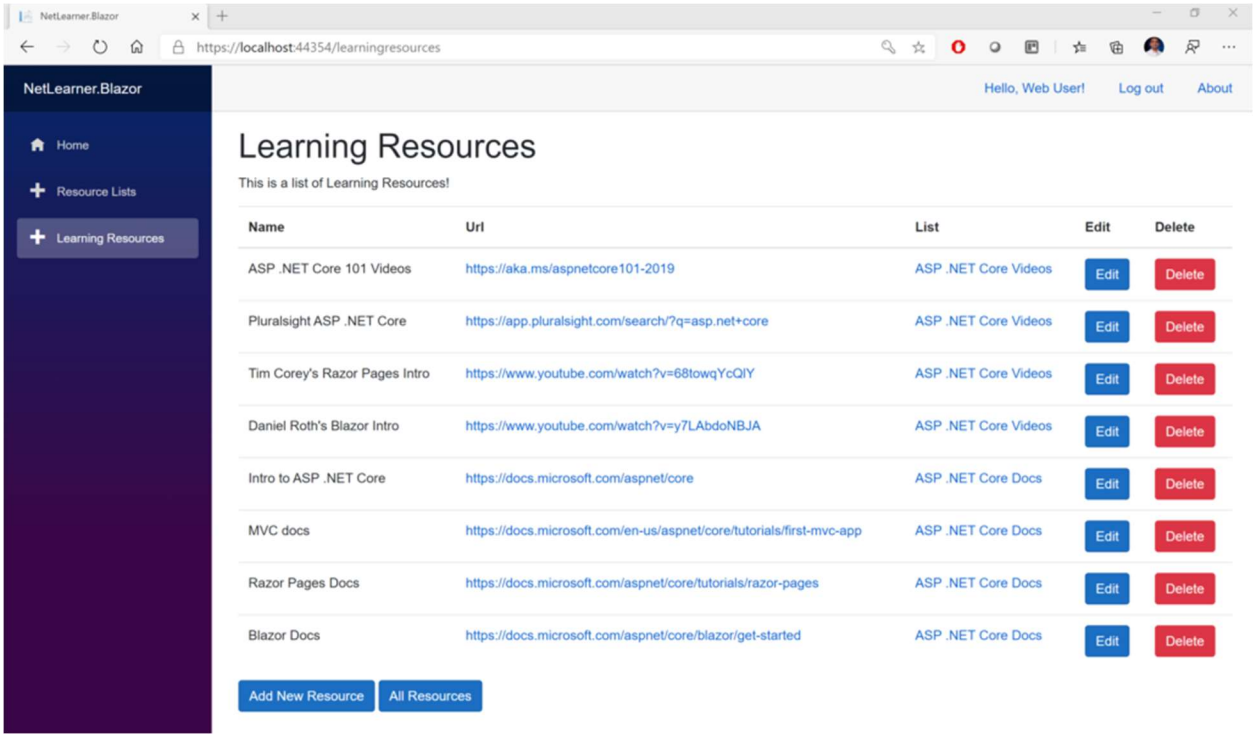
- Repository: https://github.com/shahedc/NetLearnerApp
- v0.2-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.2-alpha

In this Article:

B is for Blazor Full-Stack Web Dev

In my 2019 A-Z series, I covered Blazor for ASP .NET Core while it was still experimental. As of ASP .NET Core 3.1, server-side Blazor has now been released, while client-side Blazor (currently in preview) is expected to arrive in May 2020. This post will cover server-side Blazor, as seen in NetLearner.

To see the code in action, open the solution in Visual Studio 2019 and run the NetLearner.Blazor project. All modern web browsers should be able to run the project.



NetLearner.Blazor web app in action

Entry Point and Configuration

Let's start with Program.cs, the entry point of the 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 Generic 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 **UseStartup**() 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 IHostBuilder CreateHostBuilder(string[] args) =>
      Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
         webBuilder.UseStartup<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 call to **AddServerSideBlazor**() in **ConfigureServices**() and a call to **MapBlazorHub**() in **Configure**() while setting up endpoints with **UseEndPoints**().

```
public class Startup
{
   public void ConfigureServices(IServiceCollection services)
   {
      ...
      services.AddServerSideBlazor();
      ...
   }

   public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
   {
      ...
      app.UseEndpoints(endpoints =>
      {
         endpoints.MapControllers();
         endpoints.MapBlazorHub();
         endpoints.MapFallbackToPage("/_Host");
      });
   }
}
```

Note that the **Configure**() method takes in an app object of type **IApplicationBuilder**, similar to the IApplicationBuilder we see in regular ASP .NET Core web apps. A call to **MapFallBackToPage**() indicates the "/_Host" root page, which is defined in the _Host.cshtml page in the Pages subfolder.

Rendering the Application

This "app" (formerly defined in a static "index.html" in pre-release versions) is now defined in the aforementioned "_Host.cshtml" page. This page contains an <app> element containing the main App component.

```
<html>
...
<body>
    <app>
        <component type="typeof(App)" render-mode="ServerPrerendered"
/>
    </app>
...
</body>
</html>
```

The HTML in this page has two things worth noting: an <**app**> element within the <**body**>, and a <**component**> element of type "**App**" with a render-mode attribute set to "ServerPrerendered". This is one of 3 render modes for a Blazor component: Static, Server and ServerPrerendered. For more information on render modes, check out the official docs at:

- Blazor Hosting Models: https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-3.1

According to the documentation, this setting "*renders the component into static HTML and includes a marker for a Blazor Server app. When the user-agent starts, this marker is used to bootstrap a Blazor app.*"

The App component is defined in App.razor, at the root of the Blazor web app project. This App component contains nested authentication-enabled components, that make use of the MainLayout to display the single-page web application in a browser. If the user-requested routedata is found, the requested page (or root page) is displayed. If the user-requested routedata is invalid (not found), it displays a "sorry" message to the end user.

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
DefaultLayout="@typeof(MainLayout)" />
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

The MainLayout.razor component (under /Shared) contains the following:

- NavMenu: displays navigation menu in sidebar
- LoginDisplay: displays links to register and log in/out
- _CookieConsentPartial: displays GDPR-inspired cookie message
- @Body keyword: replaced by content of layout when rendered

The _Layout.cshtml layout file (under /Pages/Shared) acts as a template and includes a call to @RenderBody to display its content. This content is determined by route info that is requested by the user, e.g. Index.razor when the root "/" is requested, LearningResources.razor when the route "/learningresources" is requested.

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

The NavMenu.razor component contains NavLinks that point to various routes. For more information about routing in Blazor, check out the official docs at:

- Routing in Blazor: https://docs.microsoft.com/en-us/aspnet/core/blazor/routing?view=aspnetcore-3.1

LifeCycle Methods

A Blazor application goes through several lifecycle methods, including both asynchronous and non-asynchronous versions where applicable. Some important ones are listed below:

- **OnInitializedAsync**() and **OnInitialized**(): invoked after receiving initial params
- **OnParametersSetAsync**() and **OnParametersSet**(): called after receiving params from its parent, after initialization
- **OnAfterRenderAsync**() and **OnAfterRender**(): called after each render
- **ShouldRender**(): used to suppress subsequent rendering of the component
- **StateHasChanged**(): called to indicate that state has changed, can be triggered manually

These methods can be overridden and defined in the **@code** section (formerly **@functions** section) of a .razor page, e.g. LearningResources.razor.

```
 @code {
    ...
    protected override async Task OnInitializedAsync()
    {
       ...
    }
    ...
}
```

Updating the UI

The C# code in LearningResources.razor includes methods to initialize the page, handle user interaction, and keep track of data changes. Every call to an event handler from an HTML element (e.g. OnClick event for an input button) can be bound to a C# method to handle that event. The **StateHasChanged**() method can be called manually to rerender the component, e.g. when an item is added/edited/deleted in the UI.

```
<div>
    <input type="button" class="btn btn-primary" value="All Resources"
@onclick="(() => DataChanged())" />
</div>

...
private async void DataChanged()
{
    learningResources = await learningResourceService.Get();
    ResourceLists = await resourceListService.Get();
```

```
    StateHasChanged();
}
```

Note that the **DataChanged**() method includes some asynchronous calls to **Get**() methods from a service class. These are the service classes in the shared library that are also used by the MVC and Razor Pages web apps in NetLearner.

Parameters defined in the C# code can be used similar to HTML attributes when using the component, including RenderFragments can be used like nested HTML elements. These can be defined in sub-components such as ConfirmDialog.razor and ResourceDetail.razor that are inside the LearningResources.razor component.

```
<ResourceDetail LearningResourceObject="learningResourceObject"
                ResourceListValues="ResourceLists"
                DataChanged="@DataChanged">
    <CustomHeader>@customHeader</CustomHeader>
</ResourceDetail>
```

Inside the subcomponent, you would define the parameters as such:

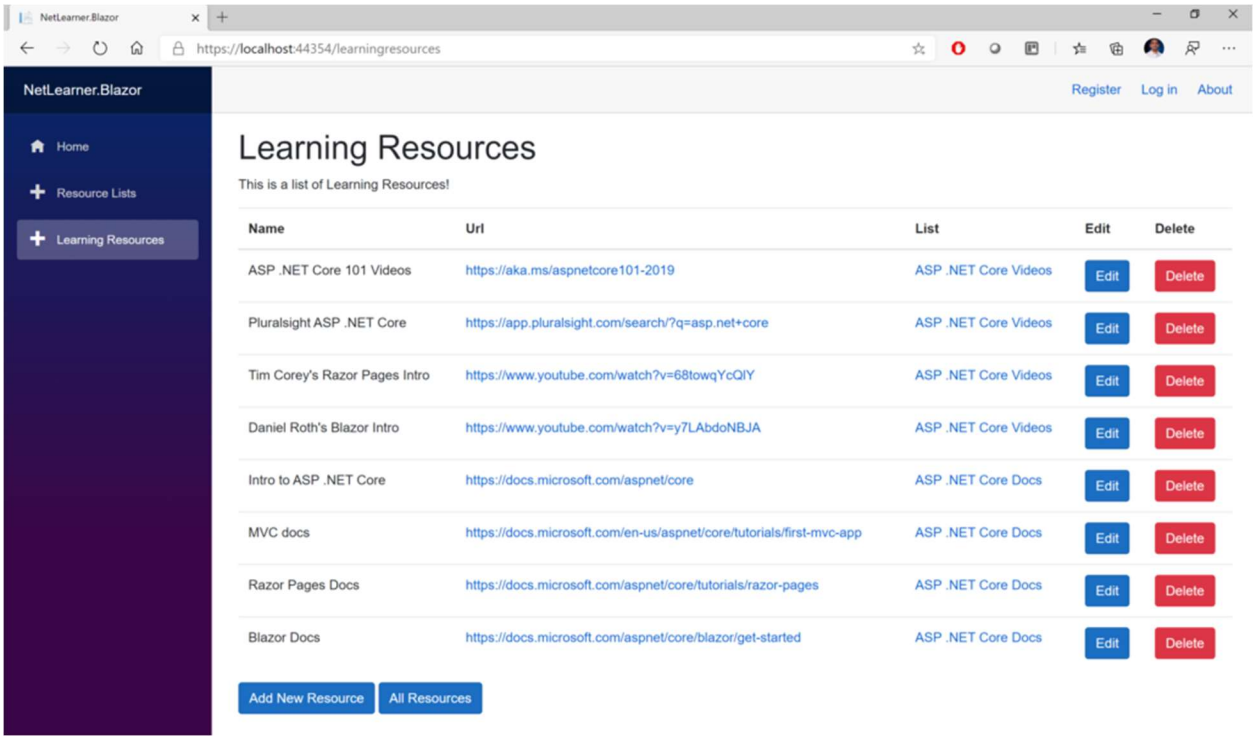
```
@code {
    [Parameter]
    public LearningResource LearningResourceObject { get; set; }

    [Parameter]
    public List<ResourceList> ResourceListValues { get; set; }

    [Parameter]
    public Action DataChanged { get; set; }

    [Parameter]
    public RenderFragment CustomHeader { get; set; }
}
```

For more information on the creation and use of Razor components in Blazor, check out the official documentation at:

- Razor Components in Blazor: https://docs.microsoft.com/en-us/aspnet/core/blazor/components?view=aspnetcore-3.1

Next Steps

Run the Blazor web app from the NetLearner repo and try using the UI to add, edit and delete items. Make sure you remove the restrictions mentioned in a previous post about NetLearner, which will allow you to register as a new user, log in and perform CRUD operations.



NetLearner.Blazor: Learning Resources

There is so much more to learn about this exciting new framework. Blazor's reusable components can take various forms. In addition to server-side Blazor (released in late 2019 with .NET Core 3.1), you can also host Blazor apps on the client-side from within an ASP .NET Core web app. Client-side Blazor is currently in preview and is expected in a May 2020 release.

References

- Official Blazor website: https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor
- Intro to Blazor: https://docs.microsoft.com/en-us/aspnet/core/blazor
- Jeff Fritz on Blazor: https://jeffreyfritz.com/2020/01/whats-old-is-new-again-web-forms-meets-blazor/
- Michael Washington's Blazor Tutorials: https://blazorhelpwebsite.com/
- Chris Sainty's Blog: https://chrissainty.com/blazor/

- Edward Charbeneau on YouTube: https://www.youtube.com/user/Backslider64/videos
- Blazor on YouTube: https://www.youtube.com/results?search_query=blazor

Cookies and Consent in ASP .NET Core 3.1

By Shahed C on January 20, 2020

Leave a reply

This is the third of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Blazor Full-Stack Web Dev in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.3-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.3-alpha

In this Article:

C is for Cookies and Consent

In this article, we'll continue to look at the (in-progress) NetLearner application, which was generated using multiple ASP .NET Core web app project (3.1) templates. In previous releases, the template made it very easy for you to store cookies and display a cookie policy. However, the latest version doesn't include cookie usage or a GDPR-compliant message out of the box.

Unless you've been living under a rock in the past couple of years, you've no doubt noticed all the GDPR-related emails and website popups since 2018. 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).

Who Moved My Cookies?

In ASP .NET Core 2.x, the standard web app project templates provided by Microsoft included GDPR-friendly popup messages, that could be accepted by the end user to set a consent cookie. As of ASP .NET

Core 3.x, this is no longer provided out of the box. However, you can still add this feature back into your project manually if needed.

Follow the instructions provided in the official docs to add this feature to your ASP .NET MVC or Razor Pages project:

- GDPR Support in ASP .NET Core: https://docs.microsoft.com/en-us/aspnet/core/security/gdpr?view=aspnetcore-3.1

But wait… how about *Blazor* web app projects? After asking a few other developers on Twitter, I decided to implement this feature myself, in my NetLearner repository. I even had the opportunity to answer a related question on Stack Overflow. Take a look at the following Blazor web project:

- NetLearner.Blazor: https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Blazor

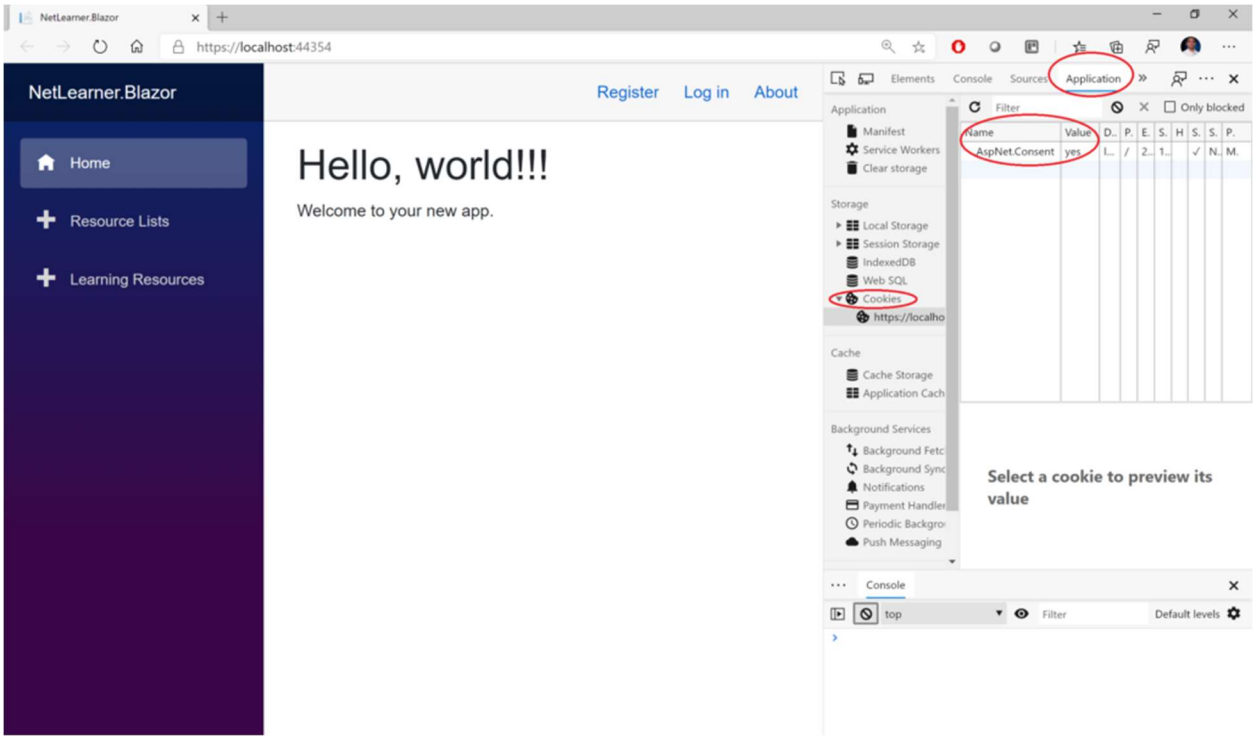
Browser Storage

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

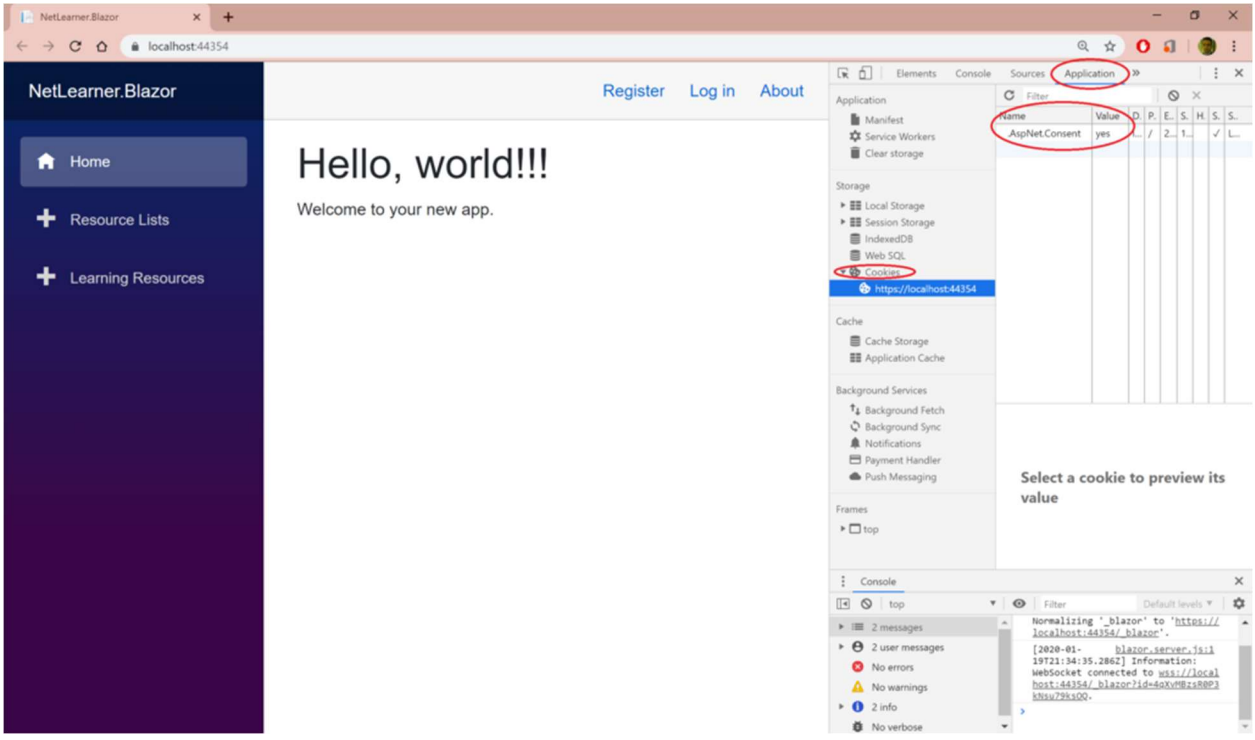
Click F12 in your browser to view the Developer Tools to see cookies grouped by website/domain.

- In (pre-Chromum) Edge/Firefox, expand Cookies under the Storage tab.
- In (Chromium-based) Edge/Chrome, expand Storage | Cookies under the Application tab .

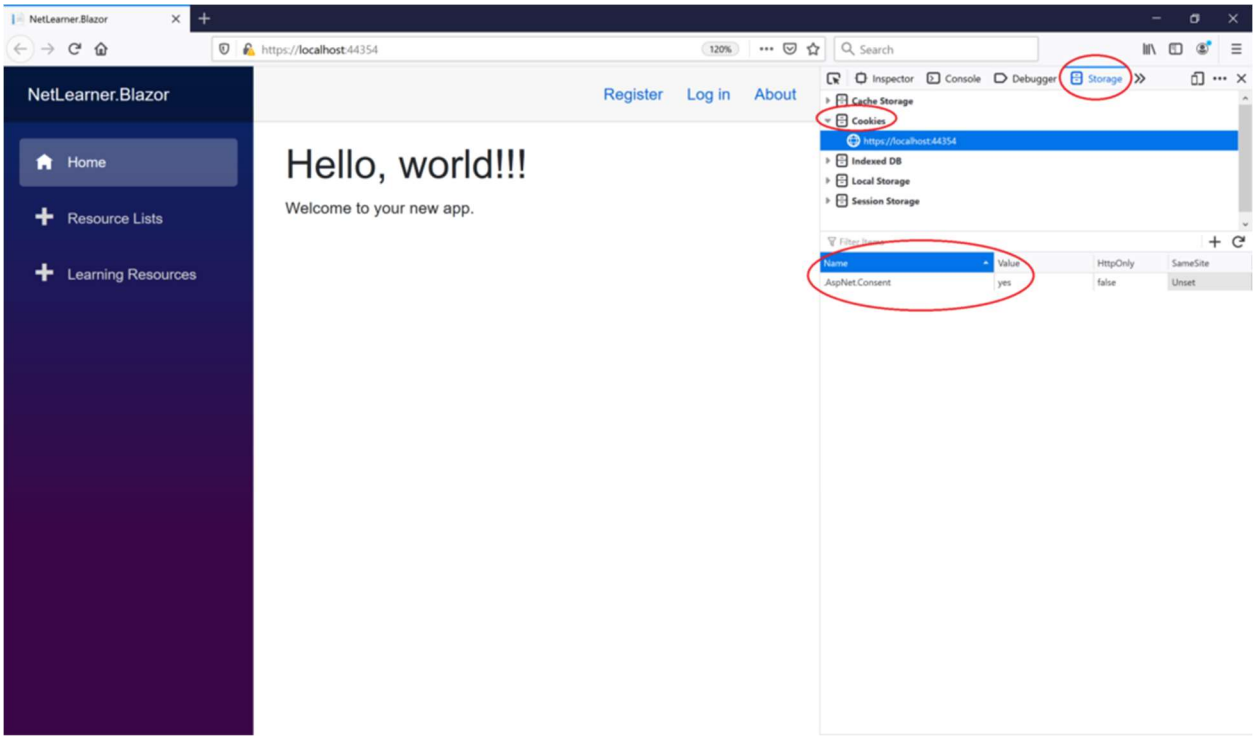
See screenshots below for a couple of examples how AspNet.Consent in stored, along with a boolean Yes/No value:



Cookies in Chromium-based Edge



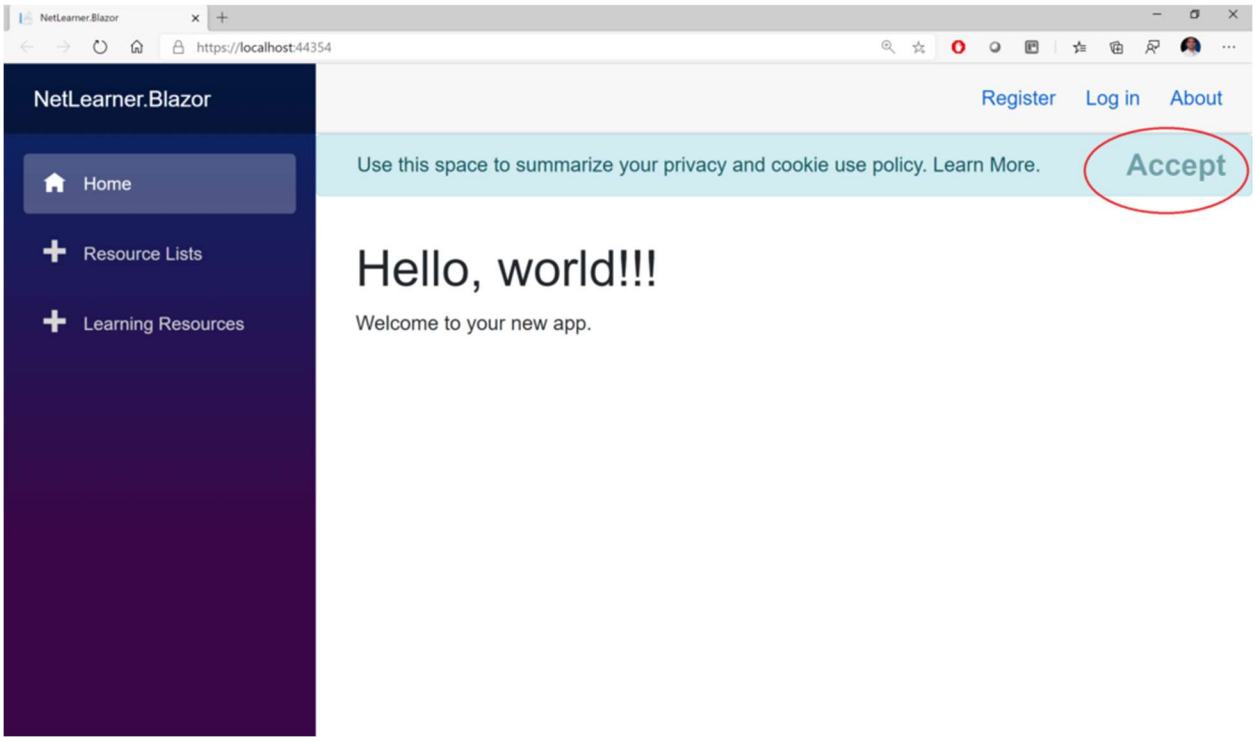
Cookies in Google Chrome



Cookies in Mozilla Firefox

Partial Views for your cookie message

The first time you launch a new template-generated ASP .NET Core 2.x web app, you should expect to see a cookie popup that appears on every page that can be dismissed by clicking Accept. Since we added it manually in our 3.x project, let's explore the code to dig in a little further.



GDPR-compliant cookie message

First, take a look at the _CookieConsentPartial.cshtml partial view in both the Razor Pages shared pages folder and the MVC shared views folder. The 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** is used to store and display a customized message for the end user.
2. This 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=Mon, 18 Jan 2021 21:55:01 GMT; path=/;
secure; samesite=none"
```

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 defined among both the Razor Pages shared pages folder and the MVC shared views folder. 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.

Blazor Implementation

Here are the steps I used in the Blazor project, tracked in the BlazorCookieExperiment branch:

1. Update ConfigureServices() in Startup.cs to set up cookie usage
2. Use JSInterop to set document.cookie in netLearnerJSInterop.js
3. Update _Host.cshtml to include the .js file
4. Observe code in _CookieConsentPartial.cshtml as reference
5. Add _CookieConsentPartial.razor component in Shared folder
6. Update MainLayout.razor to include above component

```
<div class="main">
    <div class="top-row px-4 auth">
```

```
        <LoginDisplay />
        <a href="https://docs.microsoft.com/aspnet/"
target="_blank">About</a>
    </div>

    <_CookieConsentPartial />
    <div class="content px-4">
        @Body
    </div>
</div>
```

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 may 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 (or equivalent Razor component for ther Blazor project), within the 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:

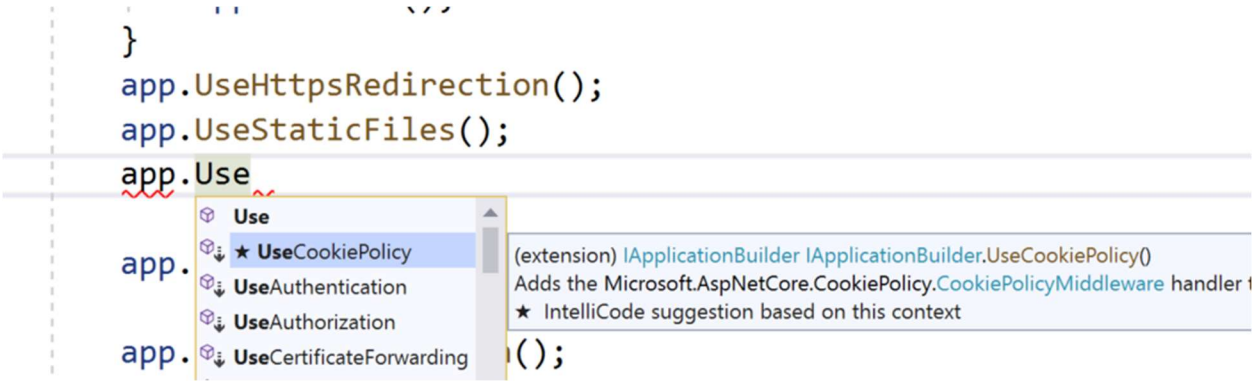
```
<a asp-area="" asp-controller="Home" asp-action="Privacy">Learn
More</a>
```

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, in the root location of Razor Pages, MVC and Blazor projects.

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

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.



app.UseCookiePolicy() in Startup.cs

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>();
```

cookieString is null if Cookie Policy disabled

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:

- Unspecified = -1
- None = 0
- Lax = 1
- Strict = 2

From the official documentation on cookie authentication, "The Cookie Policy Middleware setting for MinimumSameSitePolicy can affect the setting
of Cookie.SameSite in CookieAuthenticationOptions settings. For more information, check out the documentation at:

- Cookie Authentication: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/cookie?view=aspnetcore-3.1
- SameSiteMode Enum: https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.samesitemode?view=aspnetcore-3.1

References

- General Data Protection Regulation (GDPR) support in ASP.NET Core: https://docs.microsoft.com/en-us/aspnet/core/security/gdpr?view=aspnetcore-3.1
- Use cookie authentication without ASP.NET Core Identity: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/cookie?view=aspnetcore-3.1
- ITrackingConsentFeature Interface (Microsoft.AspNetCore.Http.Features): https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.features.itrackingconsentfeature?view=aspnetcore-3.1
- 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
- HTML DOM querySelector() Method: https://www.w3schools.com/jsref/met_document_queryselector.asp

Deploying ASP .NET Core 3.1 to Azure App Service

By Shahed C on January 27, 2020

6 Replies

This is the fourth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Cookies and Consent in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.4-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.4-alpha

In this Article:

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!"

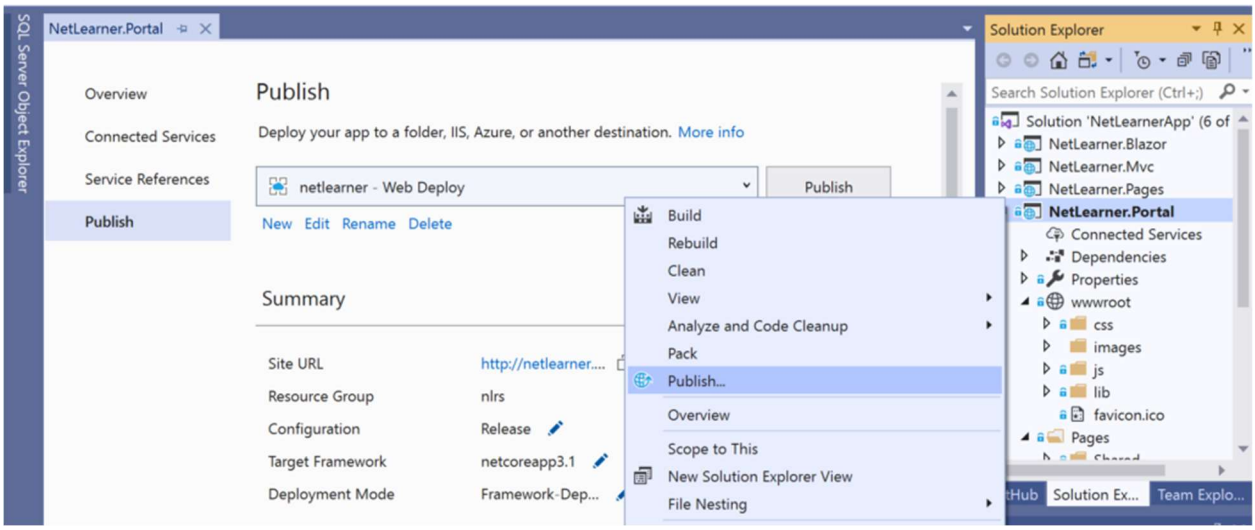
Well, there is a time and place for right-click publish. There have been many debates on this, so I won't go into the details, but here are some resources for you to see what others are saying:

- [**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/

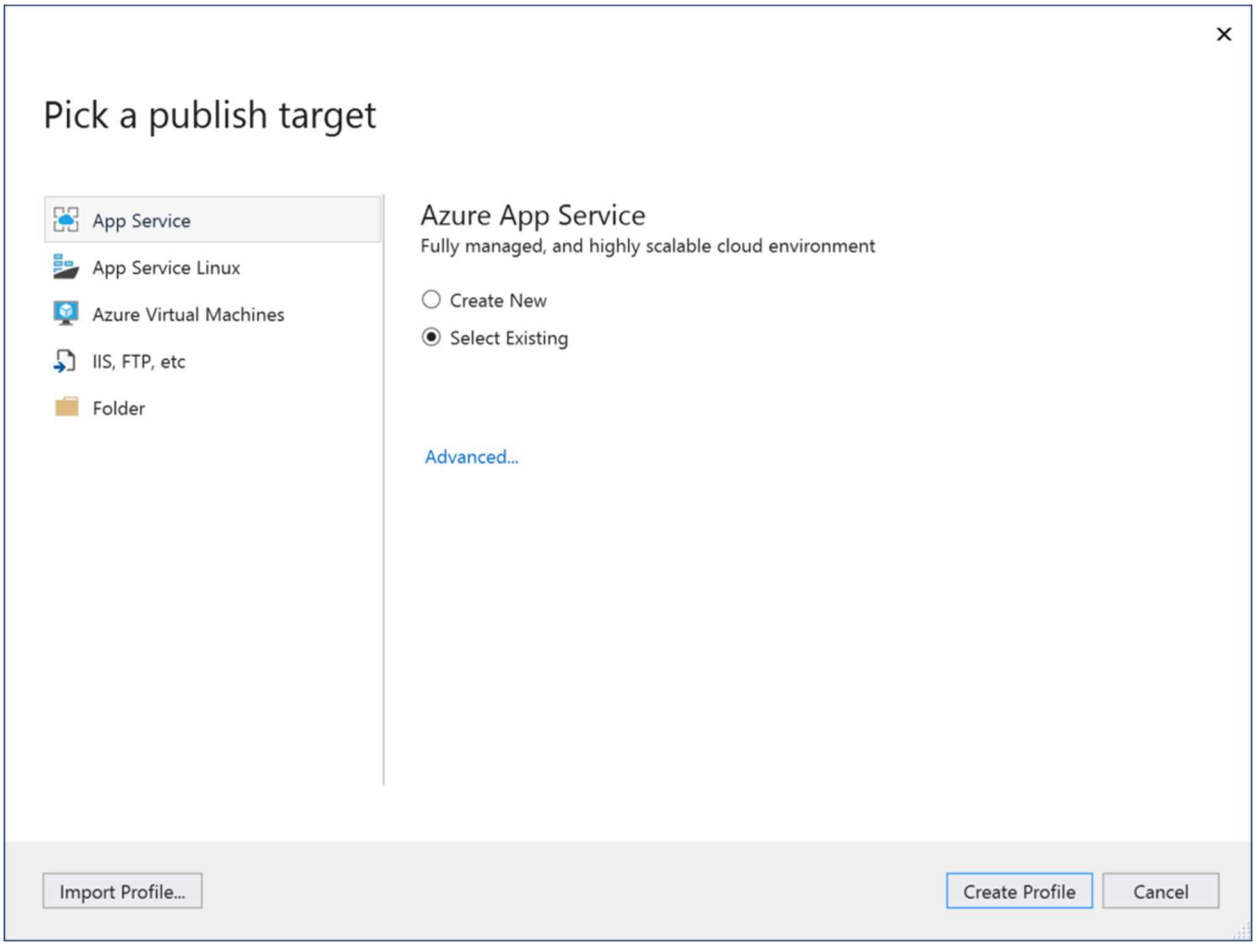
So, what's a web developer to do? To quote from the aforementioned MSDN article, *"Continuing with the theme of prototyping and experimenting, right click publish is the perfect way for existing Visual Studio customers to evaluate Azure App Service (PAAS). By following the right click publish flow you 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.



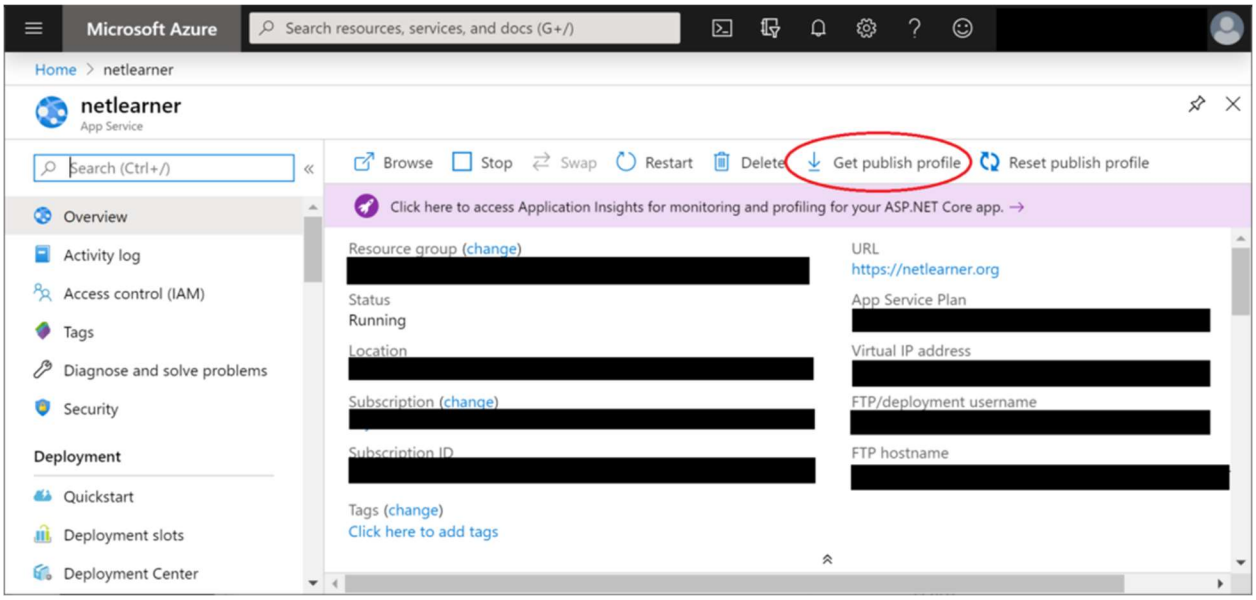
Right-click, Publish from Solution Explorer



Pick a Publish Target

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:



Get Publish Profile

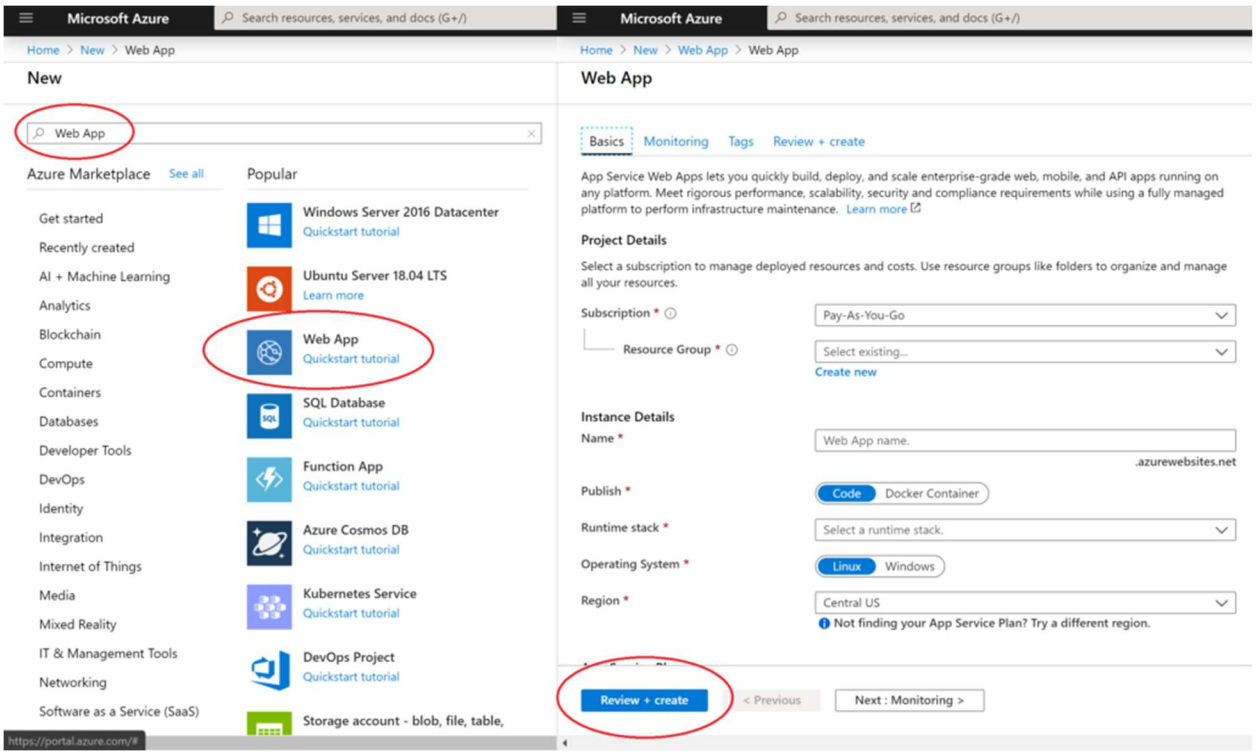
If you want to create a new Web App in Azure starting with the Azure Portal, follow the instructions below:

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:

- Subscription (select a subscription)
- Resource Group (create or use existing to group resources logically)
- Web App name (enter a unique name)
- Publish (Code or Docker Image)
- Runtime stack (.NET Core)

- App Service Plan (create or use existing to set location and pricing tier)
- OS (Windows or Linux)
- Region (e.g. East US)

5. Click Next through the screens and then click the Create button to complete the creation of your new Web App.



Create New Web App

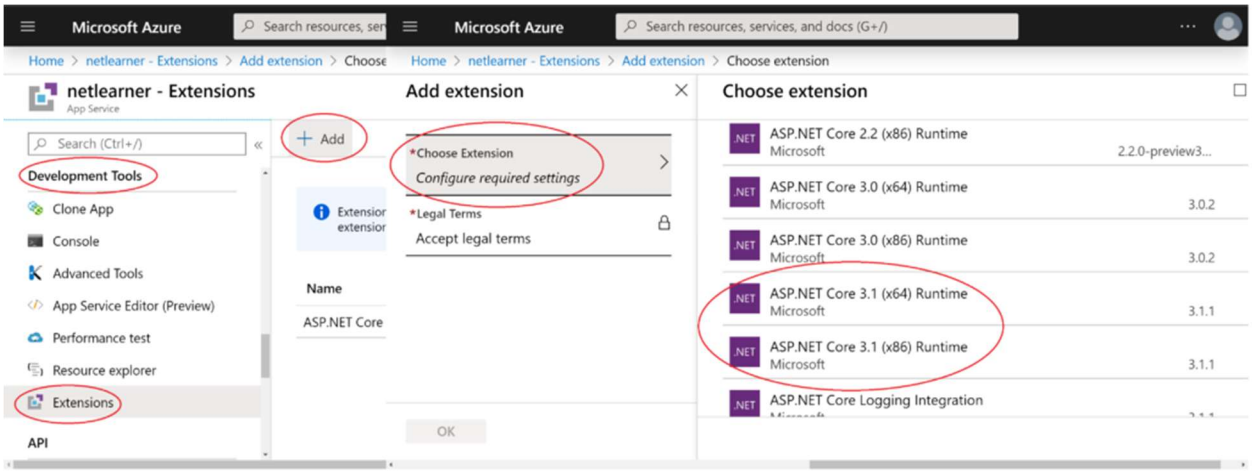
Now you can deploy to this Web App using any method you choose.

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 3.1, which is already available on Azure. If you're looking for future preview releases, 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.



Add Extension

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 covered in a future blog post in this series.

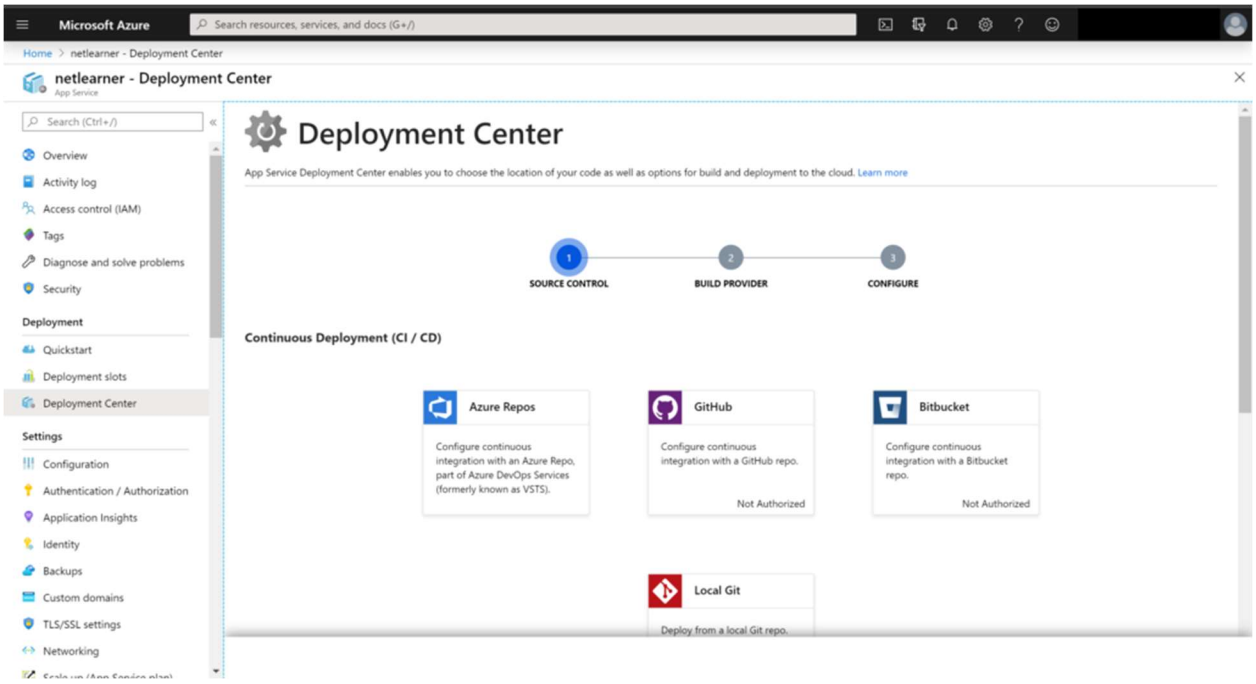
If you're interested in Deployment Slots right now, check out the official docs at:

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

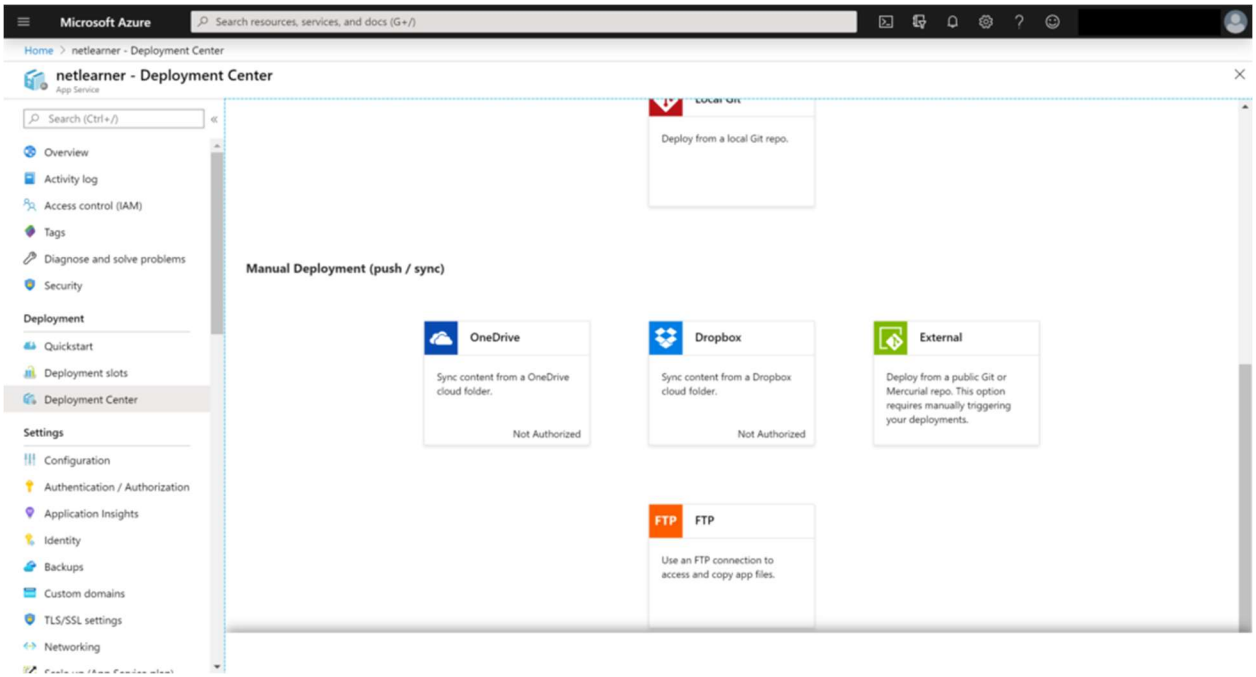
Deployment Center

In the list of features for your Web App, you will find an option to open up the new Deployment Center. Note that this has replaced the old Deployment Options. Let's go over each of these options:

1. Azure Repos
2. Github
3. Bitbucket
4. Local Git
5. OneDrive
6. Dropbox
7. External
8. FTP

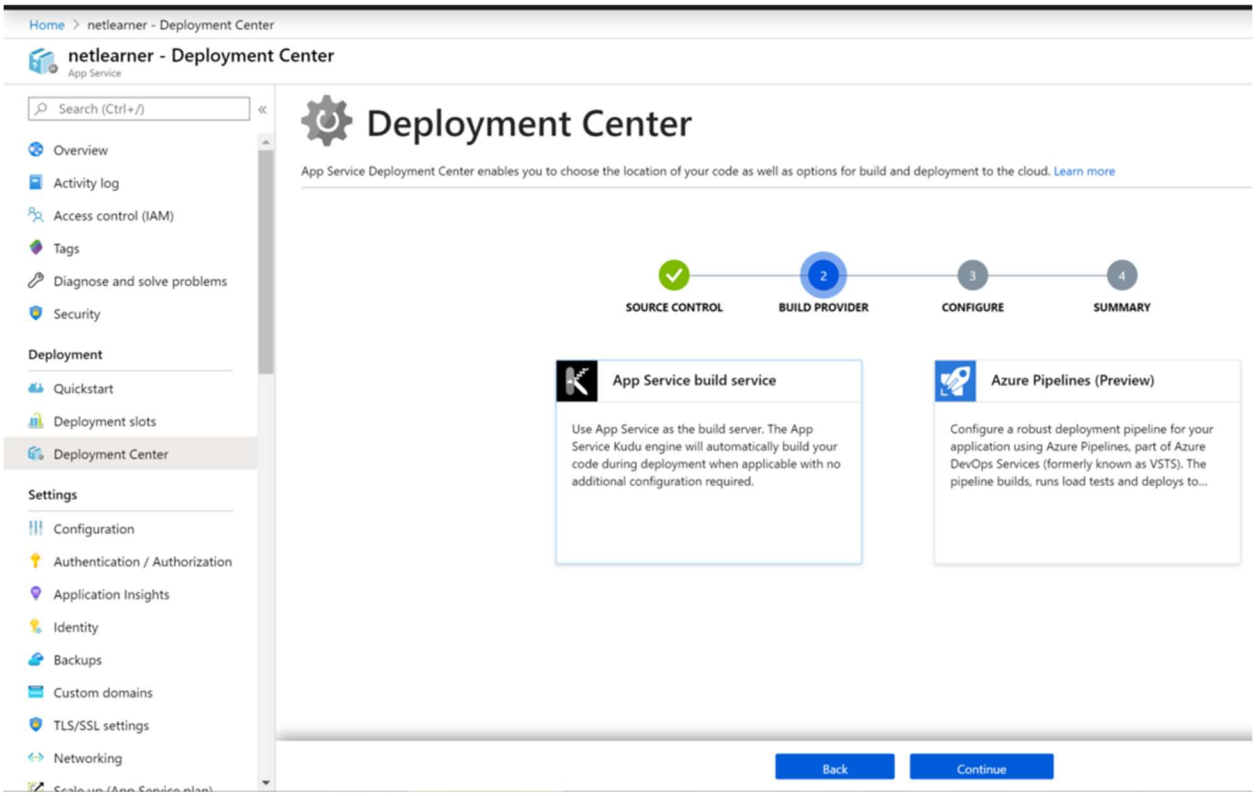


Deployment Options in Deployment Center



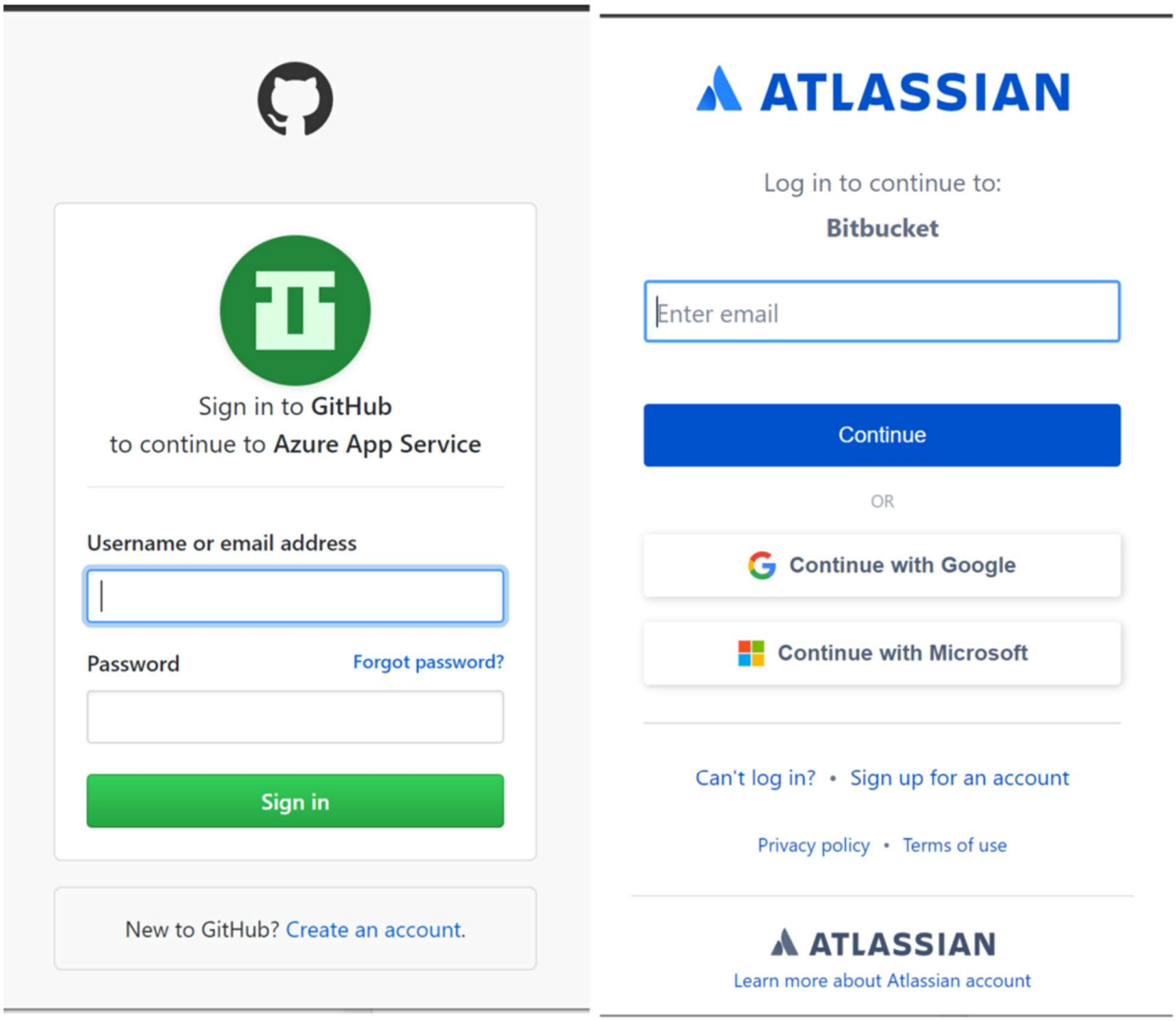
More Deployment Options

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.



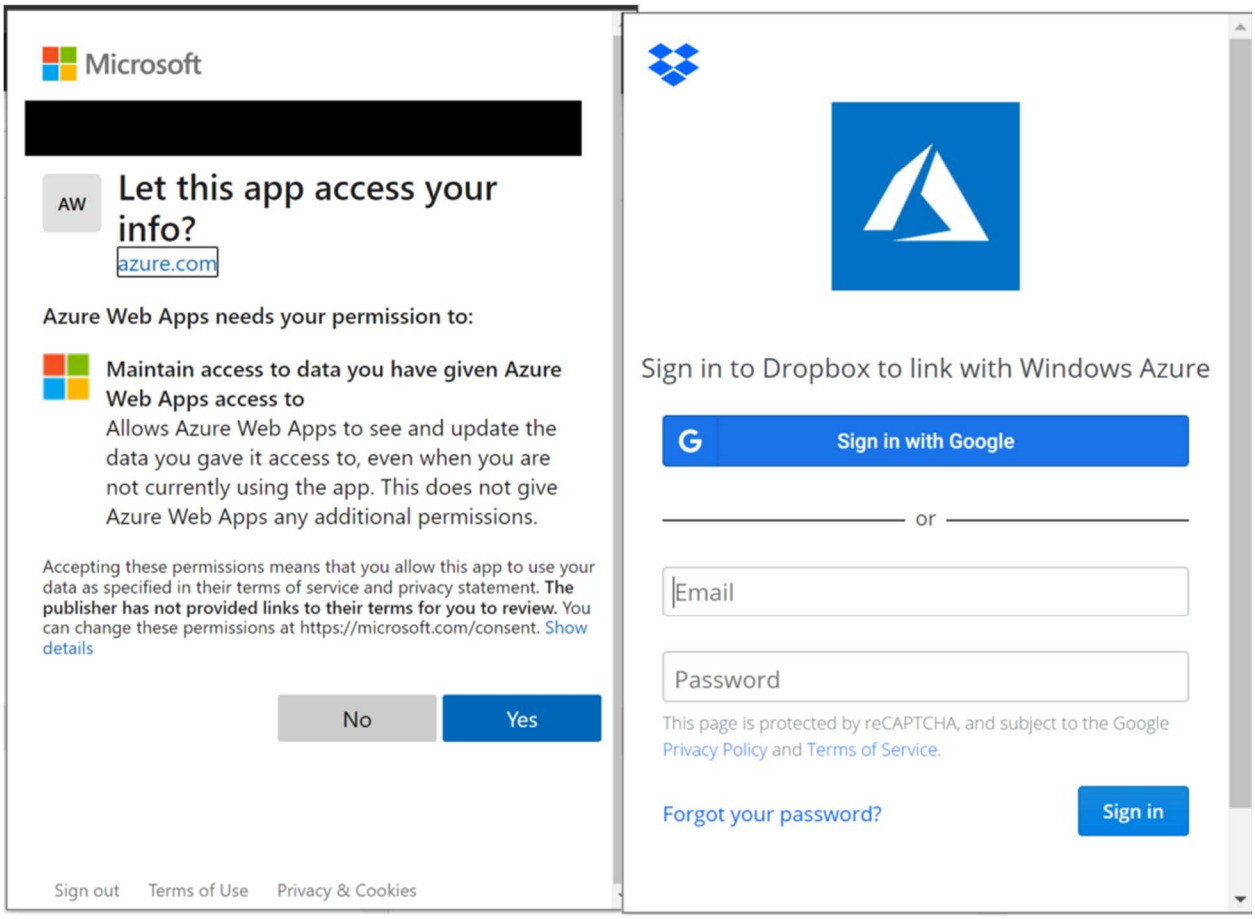
Azure Repos choices: Kudu or Pipelines?

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.



Authorize Github/Bitbucket

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.



Authorize OneDrive/DropBox

You may also select an **External** repo or **FTP** source. To learn more about **Azure Repos** and **Azure Pipelines**, check out the official docs:

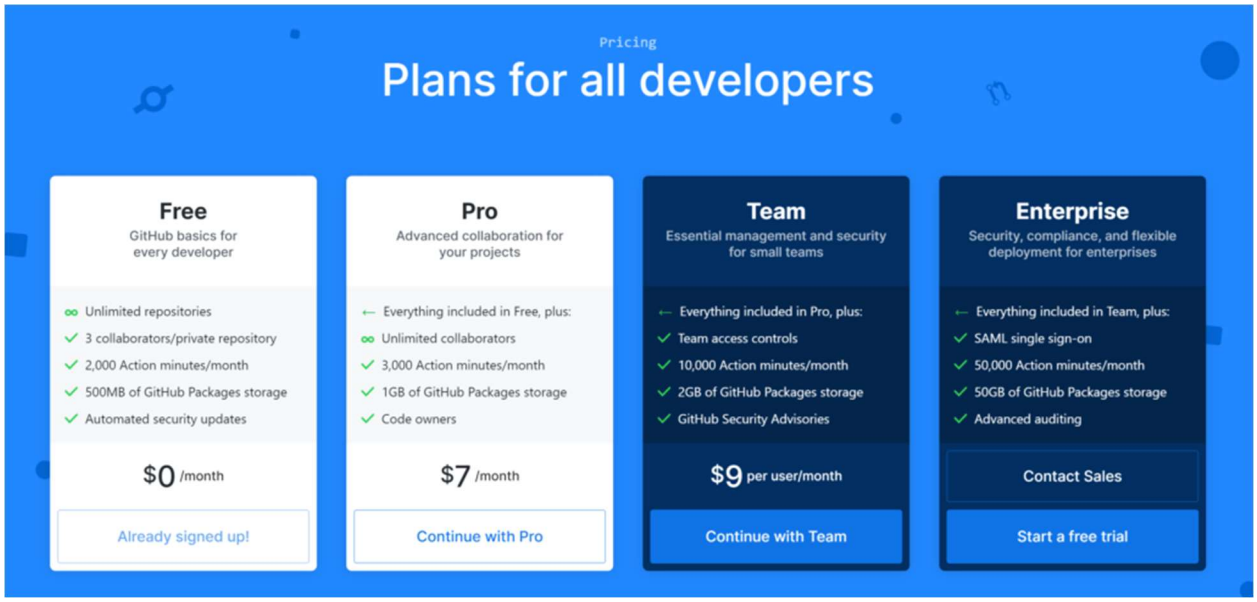
- Azure Pipelines: https://docs.microsoft.com/en-us/azure/devops/pipelines
- Overview: https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started
- Get Started: https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started-yaml
- Via Portal: https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started-azure-devops-project

GitHub Repos (includes FREE option!)

If you've been using GitHub for public open-source projects or private projects on paid accounts, now is a great to time to create private repositories for free! In 2019, GitHub started 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



GitHub pricing: Free vs Pro, Team and Enterprise

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 Sevice 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), P1V2(Premium V2 Small), PC2 (Premium Container Small), PC3 (Premium Container Medium), PC4 (Premium Container Large), I1 (Isolated Small), I2 (Isolated Medium), I3 (Isolated 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

Azure DevOps and YAML

Wait, what about Azure DevOps and YAML and Pipelines?

Since this is an A-Z series, you will have to wait for "Y is for YAML" to get more detailed information on constructing your build pipeline using YAML in Azure DevOps. If you can't wait that long, feel free to check out the following .yml sample I uploaded for use with an ASP .NET Core 3.1:

- YAML for Azure DevOps: https://github.com/shahedc/YamlForAzureDevOps

If you're reading this after June 2020, simply jump right to the "Y is for YAML" post in the 2020 A-Z series.

BONUS: for Azure SQL Database Deployment, watch the following video on MSDN Channel 9:

- SQL DB Deployment: https://channel9.msdn.com/Shows/Data-Exposed/Deployment-options-for-Azure-SQL-Database

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=azure-devops
- Continuous deployment – Azure App Service: https://docs.microsoft.com/en-us/azure/devops/pipelines/apps/cd/deploy-webdeploy-webapps?view=azure-devops
- 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=azure-devops
- Get started with Azure Pipelines: https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/index?view=azure-devops
- Continuous deployment: https://docs.microsoft.com/en-us/azure/app-service/deploy-continuous-deployment

EF Core Relationships in ASP .NET Core 3.1

By Shahed C on February 3, 2020

5 Replies

This is the fifth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Deploying ASP .NET Core 3.1 to Azure App Service

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.5-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.5-alpha

In this Article:

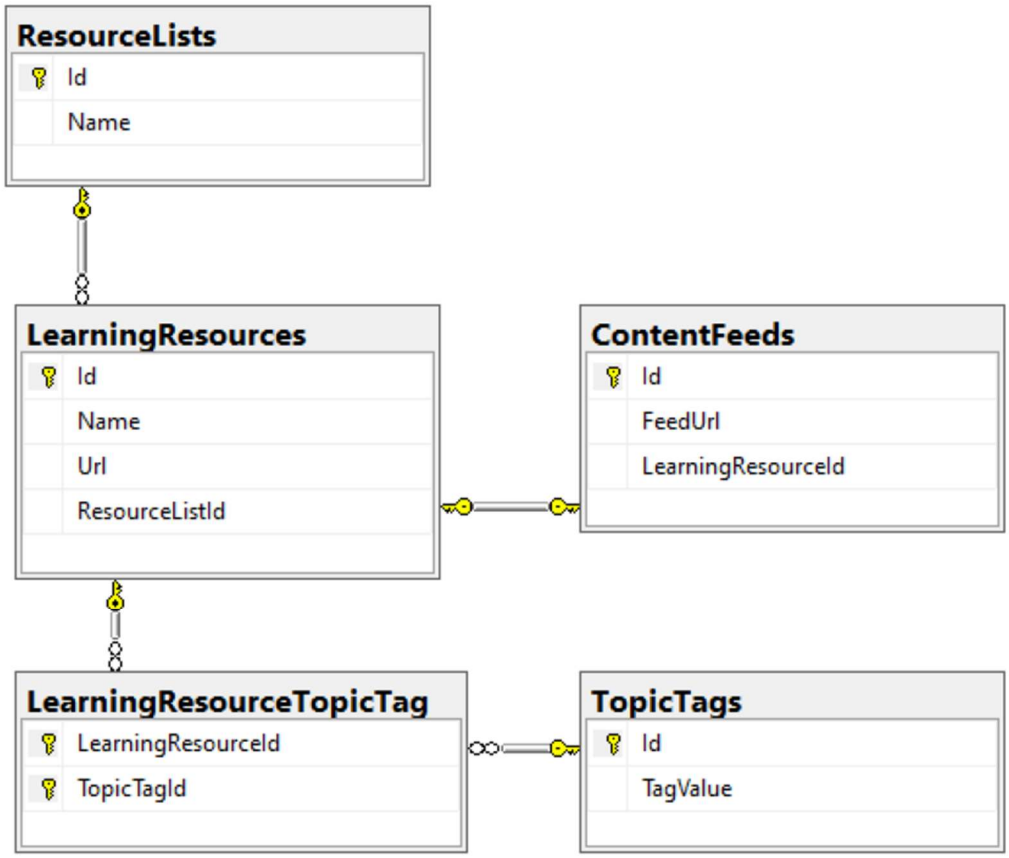
- E is for EF Core Relationships
- Classes and Relationships
- One to One
- One to Many
- Many to Many
- References

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 newer 2020 NetLearner project, to identify entities represented by C# model classes and the relationships between them.

- NetLearner on GitHub: https://github.com/shahedc/NetLearnerApp

NOTE: Please note that NetLearner is a work in progress as of this writing, so its code is subject to change. The UI web apps 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
{
    public int Id { get; set; }

    [DisplayName("Resource")]
    public string Name { get; set; }


    [DisplayName("URL")]
```

```
    [DataType(DataType.Url)]
    public string Url { get; set; }

    public int ResourceListId { get; set; }
    [DisplayName("In List")]
    public ResourceList ResourceList { get; set; }

    public ContentFeed ContentFeed { get; set; }

    public List<LearningResourceTopicTag> LearningResourceTopicTags {
get; set; }
}
```

The **ContentFeed** class represents the RSS Feed (or channel information) for an online resource, a URL that can be used to retrieve more information about the online resource, if available.

```
public class ContentFeed
{
    public int Id { get; set; }

    [DisplayName("Feed URL")]
    public string FeedUrl { get; set; }

    public int LearningResourceId { get; set; }
    public LearningResource LearningResource { get; set; }
}
```

The ResourceList 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 ResourceList
{
    public int Id { get; set; }

    public string Name { get; set; }

    public List<LearningResource> LearningResources { get; set; }
}
```

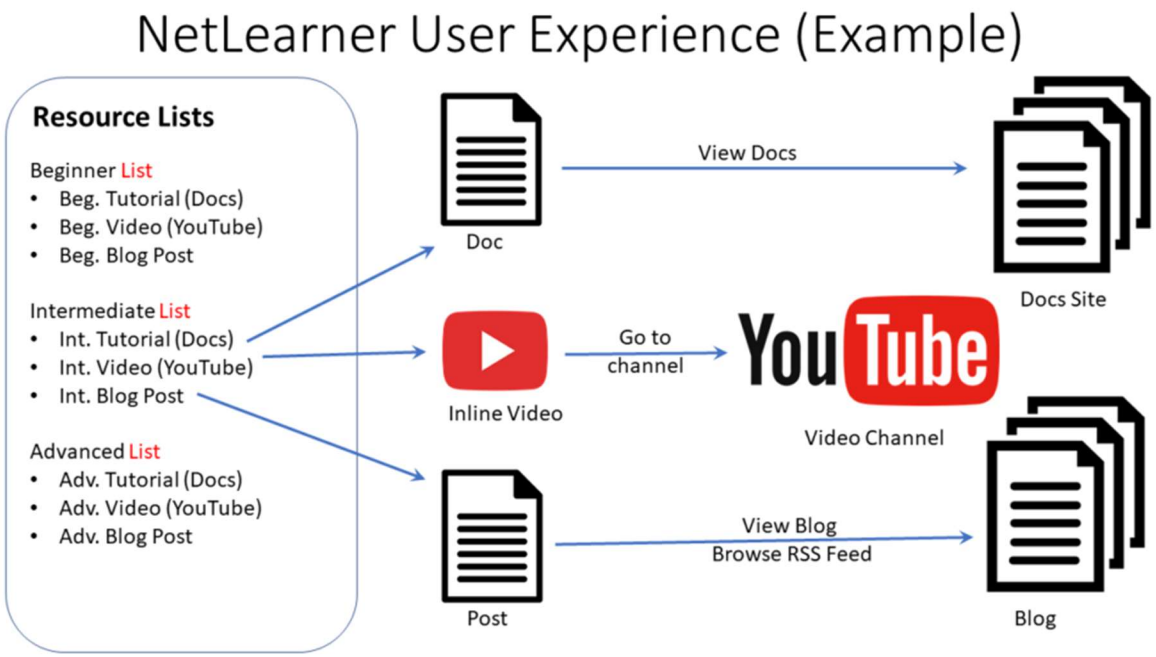
The TopicTag class represents a single “tag” value that can be used to categorize online resources. Possibly values could be “.NET Core”, “ASP.NET Core” and so on.

```
public class TopicTag
{
    public int Id { get; set; }

    [DisplayName("Tag")]
    public string TagValue { get; set; }

    public List<LearningResourceTopicTag> LearningResourceTopicTags {
```

```
get; set; }
}
```

At this point, you may have noticed both the **LearningResource** and **TopicTag** classes contain a List<T> property of **LearningResourceTopicTag**. 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** (e.g. link to a doc/video) is a part of a **ResourceList**, while each **LearningResource** also has a link back to its root site, channel or RSS feed (via **ContentFeed**).



One to One

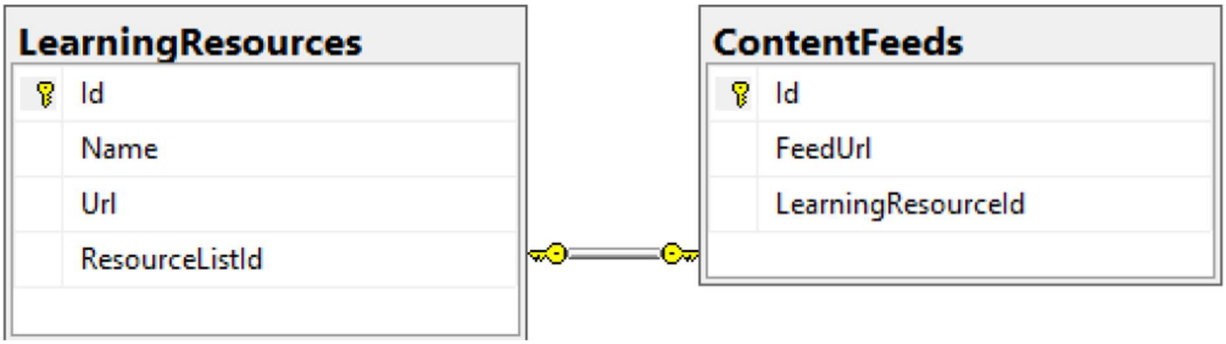
Having looked through the above entities and relationships, we can see that each **LearningResource** has a **ContentFeed**. This is an example of a 1-to-1 relationship. For example:

- Learning Resource = Wake Up and Code! blog site
- Content Feed = RSS Feed for blog site

In the two classes, we see the following code:

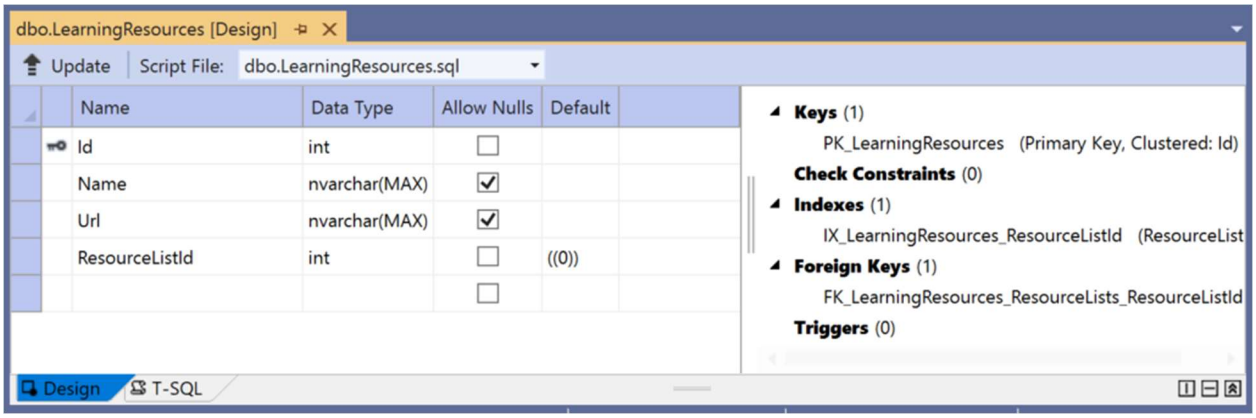
```
public class LearningResource{    public int Id { get; set; }
[DisplayName("Resource")]    public string Name { get; set; }
[DisplayName("URL")]    [DataType(DataType.Url)]    public string Url
{ get; set; }    public int ResourceListId { get; set; }
[DisplayName("In List")]    public ResourceList ResourceList { get;
set; }    public ContentFeed ContentFeed { get; set; }    public
List<LearningResourceTopicTag> LearningResourceTopicTags { get; set;
}} public class ContentFeed{    public int Id { get; set; }
[DisplayName("Feed URL")]    public string FeedUrl { get; set; }
public int LearningResourceId { get; set; }    public LearningResource
LearningResource { get; set; }}
```

Each Learning Resource has a corresponding Content Feed, so the **LearningResource** class has a property for **ContentFeed**. That's pretty simple. But in the **ContentFeed** class, you don't necessarily need a property pointing back to the **LearningResource** . In fact, all you need is a **LearningResourceId** property. EF Core will ensure that **LearningResource**.**Id** points to **ContentFeed**.**LearningResourceId** in the database. But to help with object-property navigation in your code, it is useful to include an actual **LearningResource** object in the **ContentFeed** class to point back to **LearningResource**.

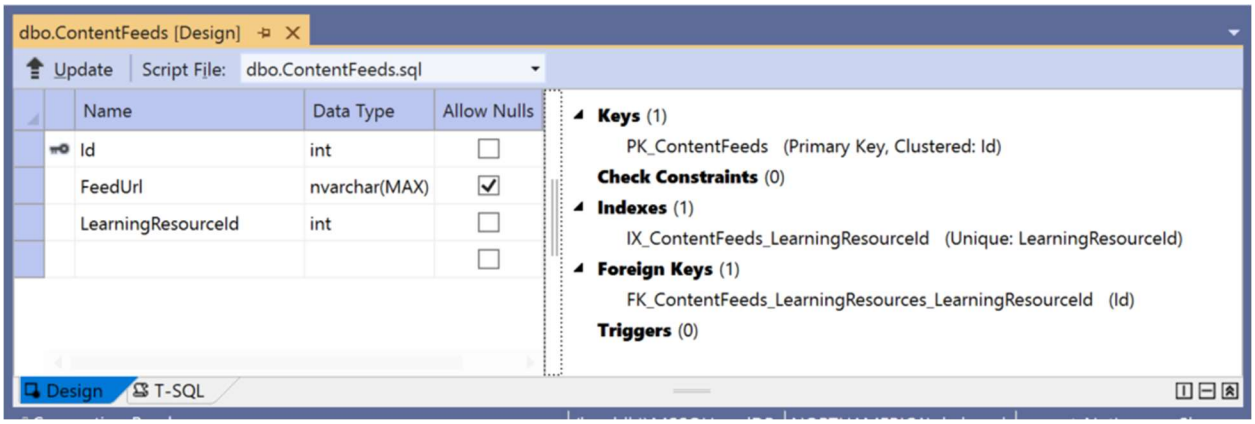


One to One Relationship

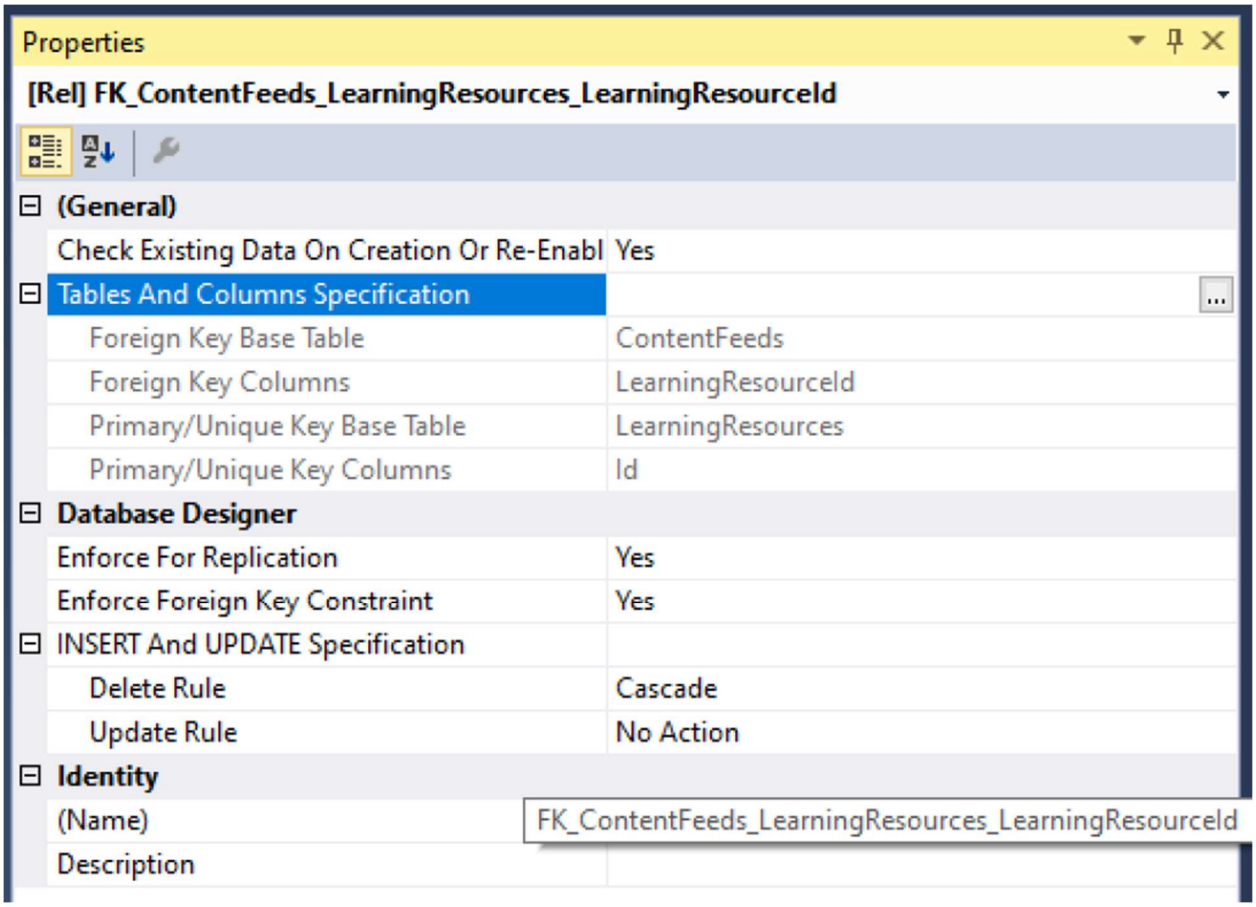
Another way of looking at the One-to-One relationship is to view the constraints of each database entity in the visuals below. Note that both tables have an Id field that is a Primary Key (inferred by EF Core) while the **ContentFeeds** table also has a Foreign Key for the **LearningResourceId** field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



LearningResources table



ContentFeeds table



One to One Relationship

One to Many

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

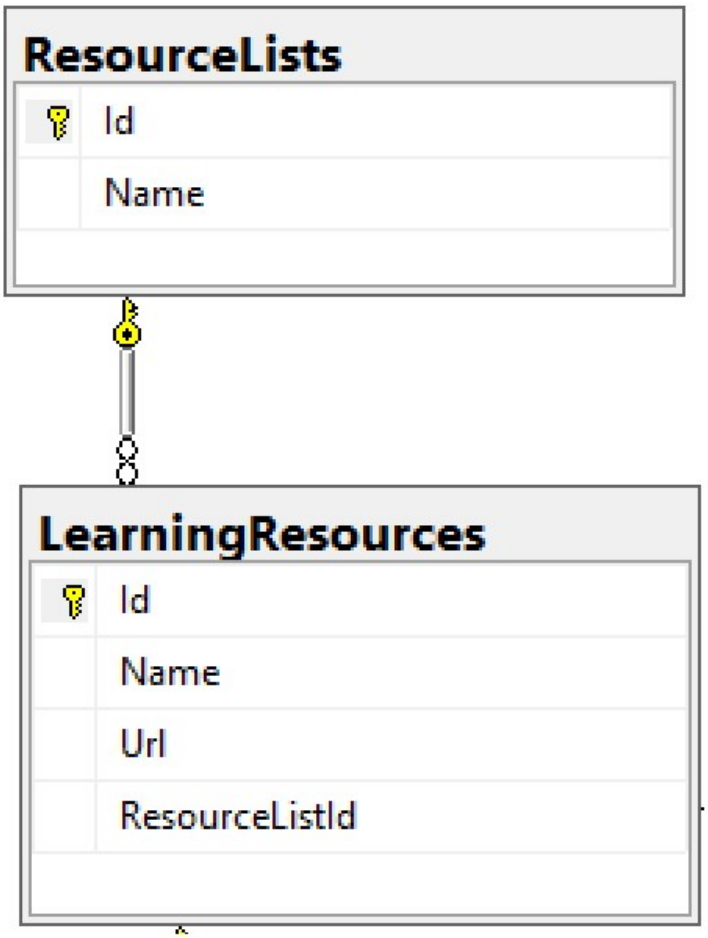
- Resource List = ASP .NET Core Blogs (parent container)
- Learning Resource = ASP .NET Core A-Z Blog Series (single URL)

In the two classes, we see the following code:

```
public class ResourceList{    public int Id { get; set; }    public
string Name { get; set; }    public List<LearningResource>
LearningResources { get; set; }} public class LearningResource{
public int Id { get; set; }    [DisplayName("Resource")]    public
string Name { get; set; }    [DisplayName("URL")]
```

```
[DataType(DataType.Url)]    public string Url { get; set; }    public
int ResourceListId { get; set; }    [DisplayName("In List")]    public
ResourceList ResourceList { get; set; }    public ContentFeed
ContentFeed { get; set; }    public List<LearningResourceTopicTag>
LearningResourceTopicTags { get; set; }}
```

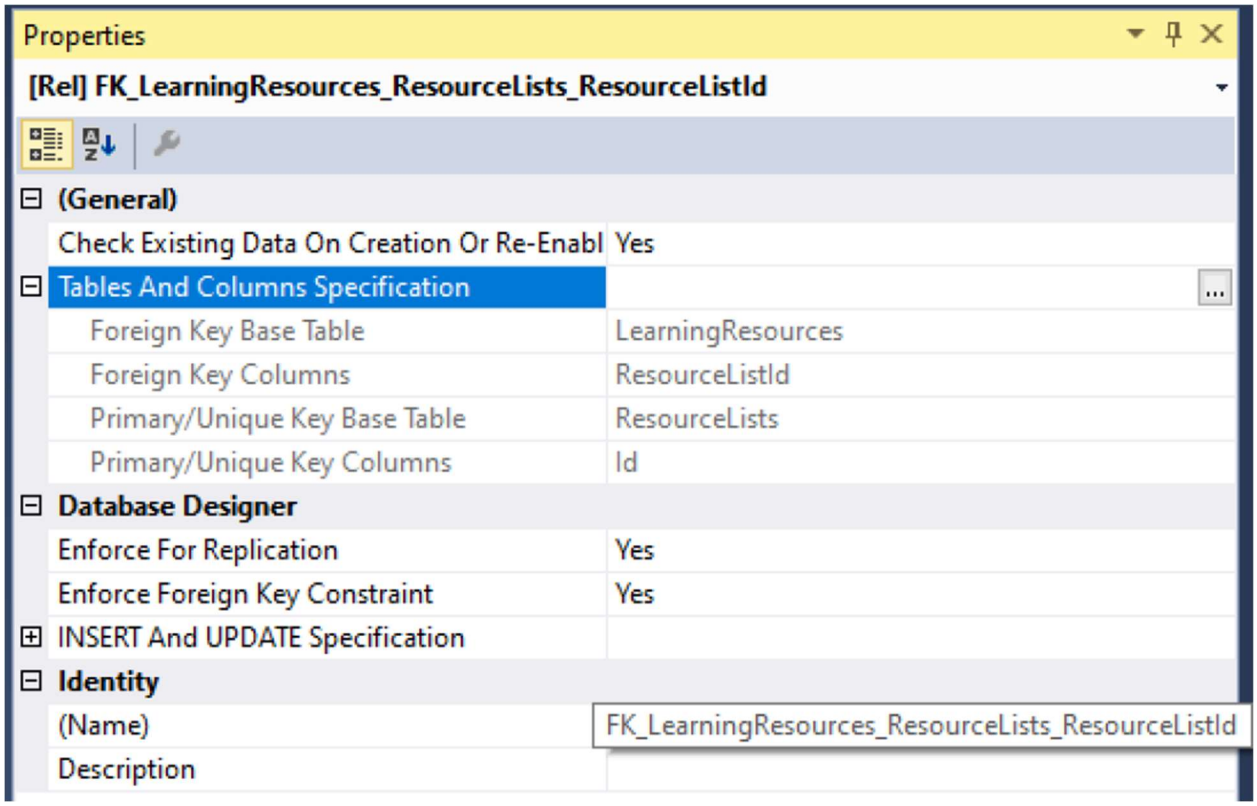
Each Resource List has zero or more Learning Resources, so the **ResourceList** class has a List<T> property for **LearningResources**. This is even simpler than the previously described 1-to-1 relationship. In the **LearningResource** class, you don't necessarily need a property pointing back to the **ResourceList**. But once again, to help with object-property navigation in your code, it is useful to include an actual **ResourceList** object in the **LearningResource** class to point back to **ResourceList**.



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 **ResourceLists** table also has a Foreign Key for the **ResourceListsId** field used for the constraint in the relationship.



One to Many Constraint

Many to Many

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

- Topic Tag = "ASP .NET Core" (tag as a text description)
- Learning Resource = Specific blog post on site (single URL)

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.

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

```
public class TopicTag
{
    public int Id { get; set; }

    [DisplayName("Tag")]
    public string TagValue { get; set; }

    public List<LearningResourceTopicTag> LearningResourceTopicTags {
get; set; }
} public class LearningResource{    public int Id { get; set; }
[DisplayName("Resource")]    public string Name { get; set; }
[DisplayName("URL")]    [DataType(DataType.Url)]    public string Url
{ get; set; }    public int ResourceListId { get; set; }
[DisplayName("In List")]    public ResourceList ResourceList { get;
set; }    public ContentFeed ContentFeed { get; set; }    public
List<LearningResourceTopicTag> LearningResourceTopicTags { get; set;
}}
```

Next, we have the LearningResourceTopicTag class as a "join entity" to connect the above:

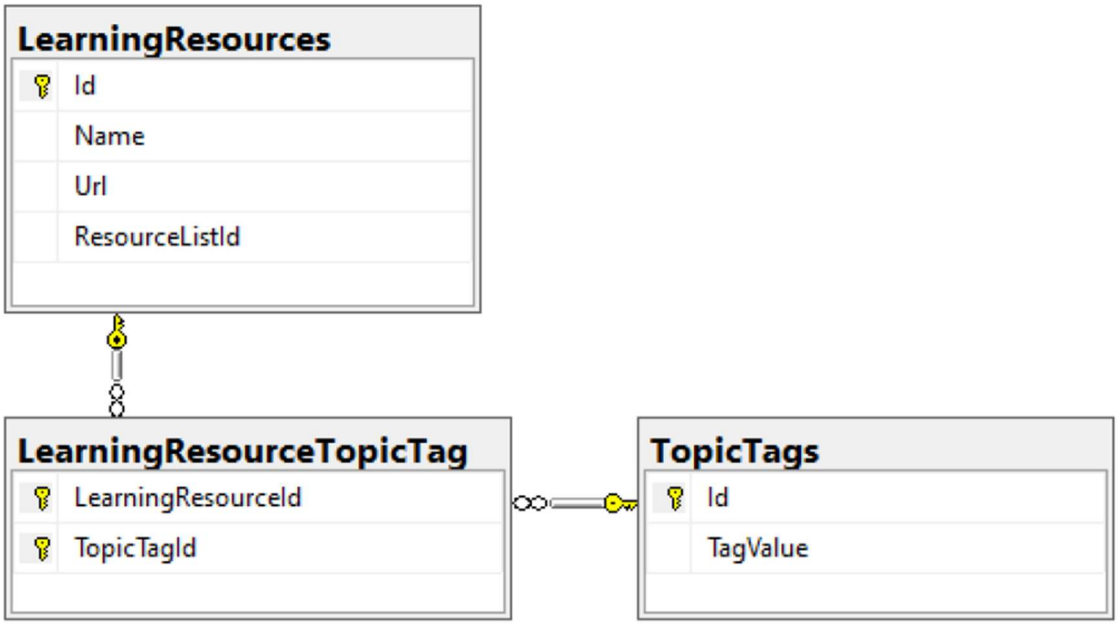
```
public class LearningResourceTopicTag
{
    public int LearningResourceId { get; set; }
    public LearningResource LearningResource { get; set; }

    public int TopicTagId { get; set; }
    public TopicTag TopicTag { 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
- **TopicTagId**: integer value, pointing back to TopicTag.Id
- **TopicTag**: optional "navigation" property, reference back to connected **TopicTag** entity

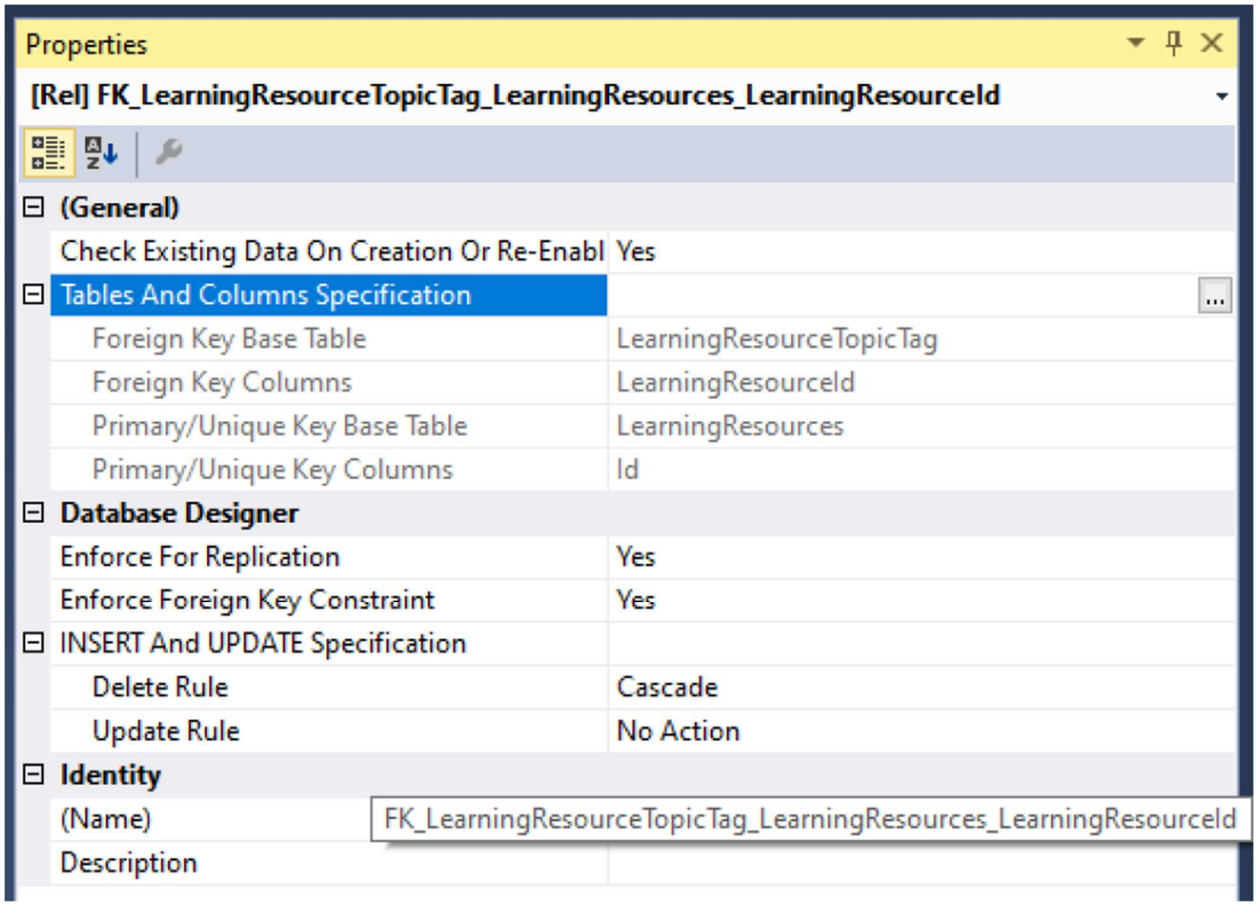
To learn more about navigation properties, check out the official docs at:

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

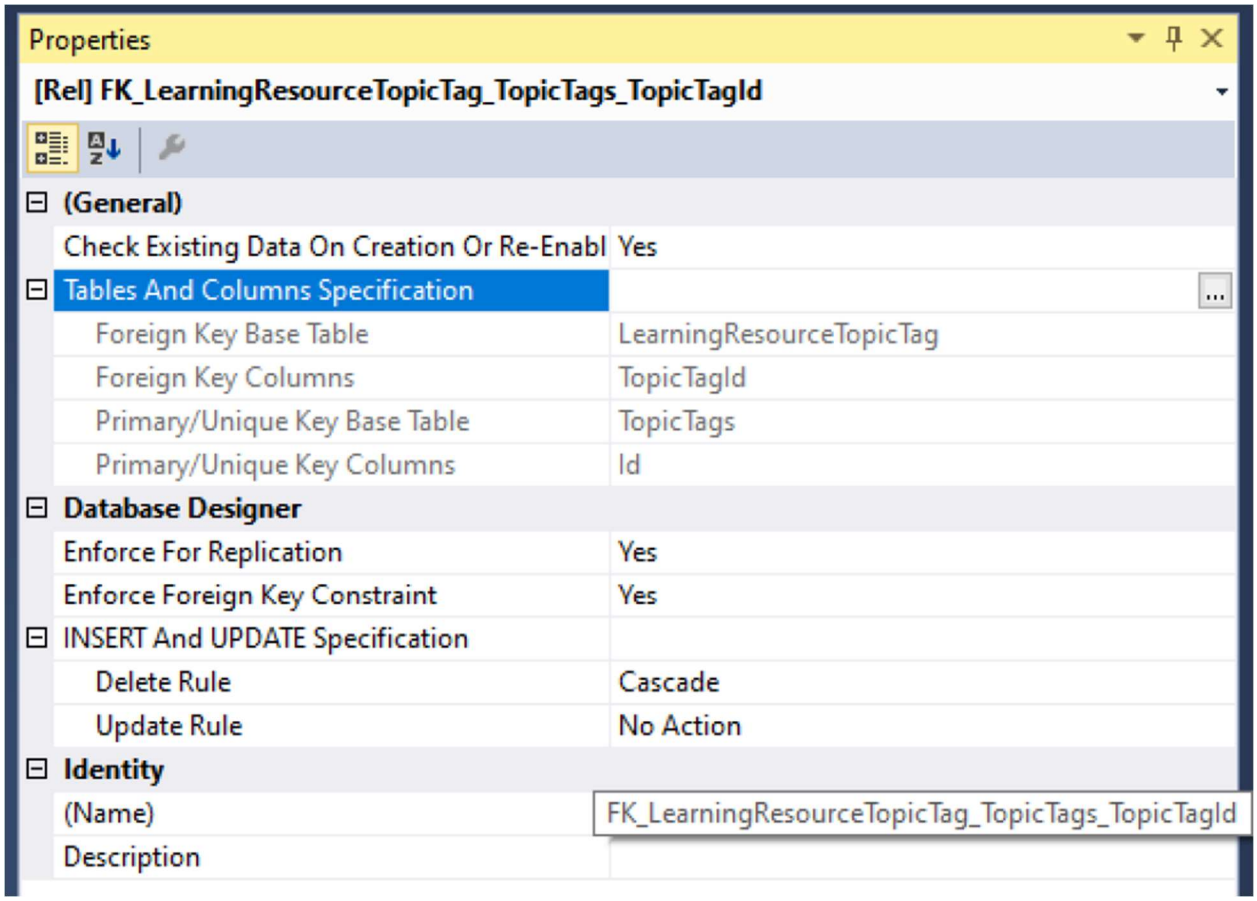


Many to Many Relationship

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 **LearningResourceTopicTag** table has a *Composite Key* for the **TopicTagId** *and* **LearningResourceId** fields used for the constraints in the relationship.



Constraints for LearningResources



Constraints for TopicTags

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

```
public class LibDbContext : IdentityDbContext
{
    ...
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        ...
        modelBuilder.Entity<LearningResourceTopicTag>()
         .HasKey(lrtt => new { lrtt.LearningResourceId, lrtt.TopicTagId
});
    }
}
```

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

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
- 3.1 Getting Started: https://www.pluralsight.com/courses/getting-started-entity-framework-core
- (For reference) 2.0 Mappings: https://www.pluralsight.com/courses/e-f-core-2-beyond-the-basics-mappings

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 3.1

By Shahed C on February 11, 2020

4 Replies

This is the sixth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- EF Core Relationships in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.6-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.6-alpha

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

- NetLearner Examples
- Razor Pages with BindProperty
- Blazor Example
- References

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 paired with 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. To avoid reusing the same id for each radio button, you could use a string array of values to concatenate a unique suffix for each radio button id.

```
// string property and value array in page model class
[BindProperty]
public string ExperienceLevel { get; set; }
public string[] ExperienceLevels = new[] { "Novice", "Beginner",
```

```
"Intermediate", "Advanced" };
<!-- input fields for radio buttons in page/view -->
@foreach (var experienceLevel in Model.ExperienceLevels)
{
    <input type="radio" asp-for="ExperienceLevel"
value="@experienceLevel"
           id="ExperienceLevel@(experienceLevel)" />
    <label for="ExperienceLevel@(experienceLevel)">
        @experienceLevel
    </label>
    <br />
}<!-- HTML generated for radio buttons -->

<input type="radio" value="Novice" id="ExperienceLevelNovice"
name="ExperienceLevel" />
<label for="ExperienceLevelNovice">
    Novice
</label>
<br />
<input type="radio" value="Beginner" id="ExperienceLevelBeginner"
name="ExperienceLevel" />
<label for="ExperienceLevelBeginner">
    Beginner
</label>
<br />
<input type="radio" value="Intermediate"
id="ExperienceLevelIntermediate" name="ExperienceLevel" />
<label for="ExperienceLevelIntermediate">
    Intermediate
</label>
<br />
<input type="radio" value="Advanced" id="ExperienceLevelAdvanced"
name="ExperienceLevel" />
<label for="ExperienceLevelAdvanced">
    Advanced
</label>
<br />
```

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

NetLearner Examples

In the NetLearner repository, you'll find multiple web projects with various views/pages and controllers where applicable. All models are shared in the SharedLib project.

- **Models**: In the "Models" folder of the shared library, you'll find a LearningResource.cs class (shown below) with some fields we will use to display HTML form elements.
- **Views**: Within the "Views" subfolder of the MVC project, the "LearningResources" subfolder contains auto-generated views for the LearningResourcesController's methods.
- **Controllers**: The LearningResourcesController class was auto-generated for the Human model, and then its functionality was extracted into a corresponding service class.
- **Services**: The LearningResourceService class was written manually to provide CRUD functionality for the LearningResourcesController, and equivalent Razor/Blazor code.
- **Razor Pages**: Within the "LearningResources" subfolder of the Razor Pages project's Pages folder, the pairs of .cshtml and .cs files make up all the CRUD functionality for LearningResources entities.
- **Blazor Components**: Within the "Pages" subfolder of the Blazor project, the LearningResources.razor component contains both its own HTML elements and event handlers for CRUD functionality. The ResourceDetail.razor component is reused by its parent component.

```
public class LearningResource
{
    public int Id { get; set; }

    [DisplayName("Resource")]
    public string Name { get; set; }


    [DisplayName("URL")]
    [DataType(DataType.Url)]
    public string Url { get; set; }

    public int ResourceListId { get; set; }
    [DisplayName("In List")]
    public ResourceList ResourceList { get; set; }

    [DisplayName("Feed Url")]
    public string ContentFeedUrl { get; set; }

    public List<LearningResourceTopicTag> LearningResourceTopicTags {
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="Name" class="control-label"></label>
    <input asp-for="Name" class="form-control" />
    <span asp-validation-for="Name" class="text-danger"></span>
</div>
... [HttpPost]
public async Task<IActionResult> Create(
Bind("Id,Name,Url,ResourceListId,ContentFeedUrl")] LearningResource
learningResource)

...
[HttpPost]
public async Task<IActionResult> Edit(
int id,
[Bind("Id,Name,Url,ResourceListId,ContentFeedUrl")] LearningResource
learningResource)
```

The Razor Pages in their respective project handle the same functionality in their own way.

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 NetLearner repo, you'll find a Razor web project with multiple subfolders, including Pages and their "code-behind" Page Model files.

- **Pages**: Within the "Pages" subfolder, the "LearningResources" subfolder within it contains Razor Pages along with corresponding .cs classes that contain the necessary Get/Post methods.
- **Page Models**: In the same LearningResources subfolder, the corresponding Page Models contain the necessary CRUD functionality. The implementation of the CRUD functionality from has been extracted into the Shared Library's corresponding LearningResourceService service class.

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

```
<div class="form-group">
    <label asp-for="LearningResource.Name" class="control-
```

```
label"></label>
    <input asp-for="LearningResource.Name" class="form-control" />
    <span asp-validation-for="LearningResource.Name" 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 LearningResource LearningResource { get; set; }
```

This [**BindProperty**] attribute allows you to declaratively bind the LearningResource 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();
    }

    await _learningResourceService.Add(LearningResource);

    var resourceList = await _resourceListService.Get();
    ViewData["ResourceListId"] = new SelectList(resourceList, "Id",
```

```
"Name", LearningResource.ResourceListId);
    return RedirectToPage("./Index");
}
```

Note that the **LearningResource** entity is passed to the service class (which passes it to the shared DB Context) to add it to the database. If you were to remove the aforementioned [**BindProperty**] attribute, LearningResource 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)]
```

Blazor Example

The Blazor version of NetLearner also reuses the same shared library for its CRUD functionality and entity models via the shared service classes. However, its front-end web app looks noticeably different. There are no controllers or views. Rather, the .razor files contain HTML elements and the C# code necessary to handle user interaction.

An overall example of the Blazor web project is explained earlier in this series:

- Blazor Full-Stack Web Dev: https://wakeupandcode.com/blazor-full-stack-web-dev-in-asp-net-core-3-1/

After the A-Z series is complete, stay tuned for new content that will explain Blazor's use of <EditForm> and the use of Input Components to render HTML elements and handle events. For now, check out the official documentation at:

- Components in Blazor: https://docs.microsoft.com/en-us/aspnet/core/blazor/components
- Blazor forms and validation: https://docs.microsoft.com/en-us/aspnet/core/blazor/forms-validation

Input Components in Blazor include the following:

- InputText: renders an <input> element
- InputTextArea: renders an <textarea> element
- InputSelect: renders a <select> element
- InputNumber: renders an <input> element of type=number
- InputCheckbox: renders an <input> element of type=checkbox
- InputDate: renders an <input> element of type=date

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.learnrazorpages.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://jonhilton.net/aspnet-core-forms-cheat-sheet/

Generic Host Builder in ASP .NET Core 3.1

By Shahed C on February 17, 2020

7 Replies

This is the seventh of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Forms and Fields in ASP .NET Core 3.1

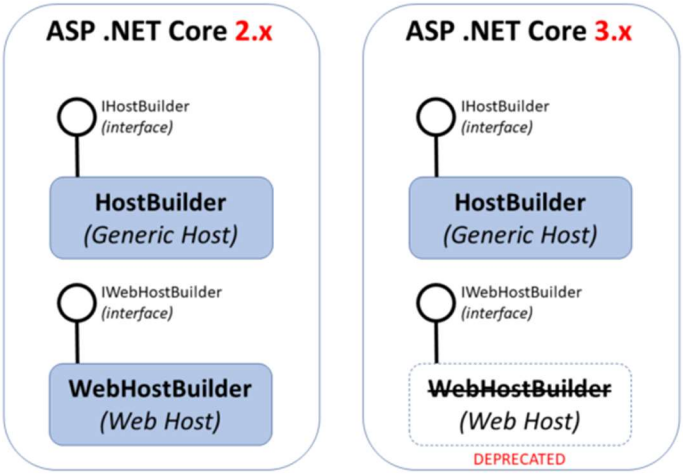
NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.7-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.7-alpha

In this Article:

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 has now replaced the Web Host Builder as of v3.0 in 2019.



Generic Host Builder in ASP .NET Core 3.x

History Lesson: Generic Host Builder in 2.x

So, if the Generic Host Builder wasn't used for web hosting in v2.x, what was it there for? The aforementioned non-HTTP workloads include a number of capabilities according to the 2.2 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();
})
```

More History: Web Host Builder in 2.x

The **WebHostBuilder** class was made 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 was used for hosting web apps in v2.x. As mentioned in the previous section, it has since been replaced by the Generic Host Builder as of v3.0. At a minimum, the **Main**() method of your ASP .NET Core 2.x web app would have looked 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 that 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

As of 2019, ASP .NET Core 3.x allows you to use the updated Generic Host Builder instead of the Web Host Builder in your web apps. The ASP .NET Core templates were updated to include the Generic Host Builder as of v3.0 Preview 2. You should use v3.1 since it's a LTS (Long-Time Support) release.

At a minimum, the **Main**() method of your .NET Core 3.1 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.x 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.x.

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** has now been *deprecated* and could be *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/
- v2.1 .NET Generic Host: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-2.2
- v3.1 .NET Generic Host: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-3.1
- 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 3.1

By Shahed C on February 24, 2020

7 Replies

This is the eighth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Generic Host Builder in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.8-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.8-alpha

In this Article:

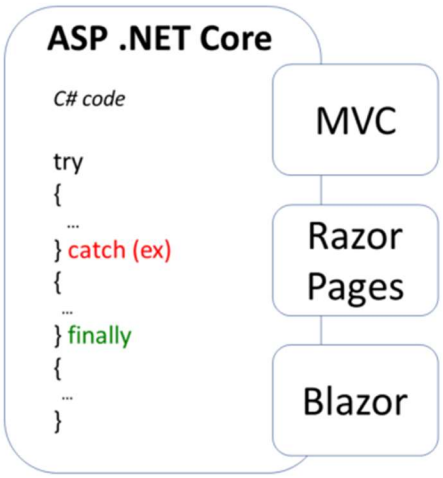
- Error Handling in Blazor
- Logging Errors
- Transient Fault Handling
- References

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.



try-catch-finally in C#

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 NetLearner

In the NetLearner sample app, the LearningResourcesController uses a LearningResourceService from a shared .NET Standard Library to handle database updates. In the overloaded Edit() method, it wraps a call to the Update() method from the service class to catch a possible DbUpdateConcurrencyException exception. First, it checks to see whether the ModelState is valid or not.

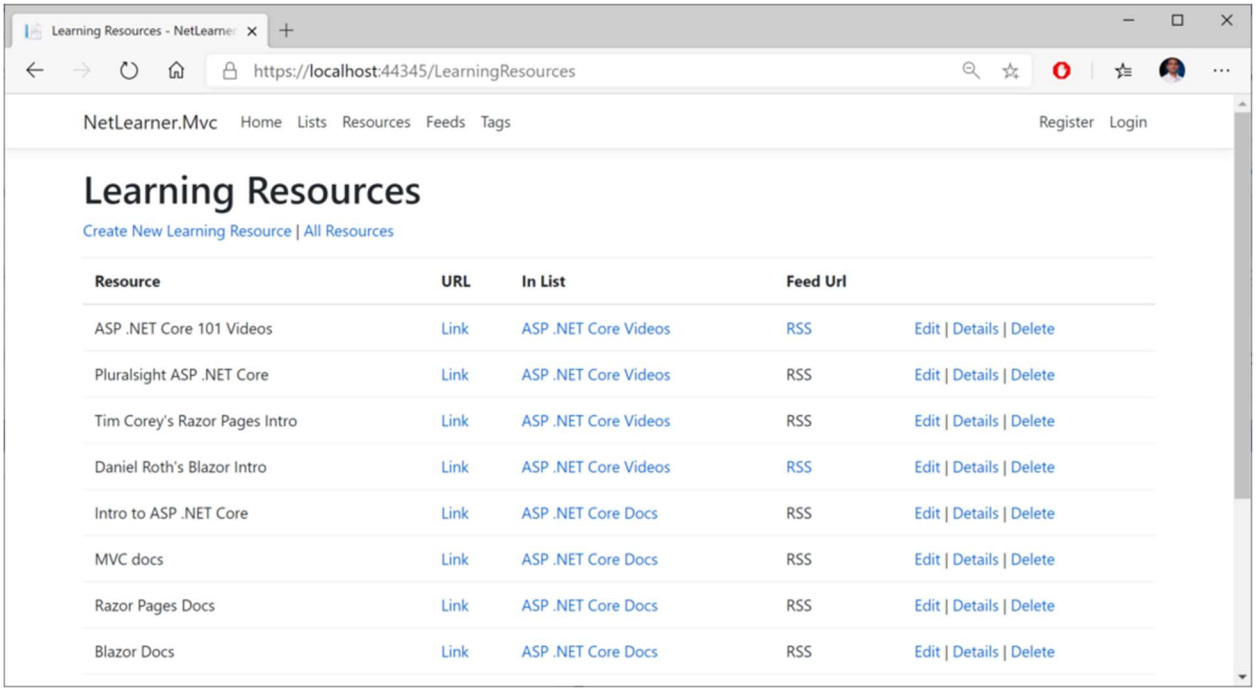
```
if (ModelState.IsValid)
{
    ...
}
```

Then, it tries to update user-submitted data by passing a LearningResource object. If a DbUpdateConcurrencyException exception occurs, there is a check to verify whether the current LearningResource even exists or not, so that a 404 (NotFound) can be returned if necessary.

```
try
{
    await _learningResourceService.Update(learningResource);
}
catch (DbUpdateConcurrencyException)
{
    if (!LearningResourceExists(learningResource.Id))
    {
        return NotFound();
    }
    else
    {
        throw;
    }
}
return RedirectToAction(nameof(Index));
```

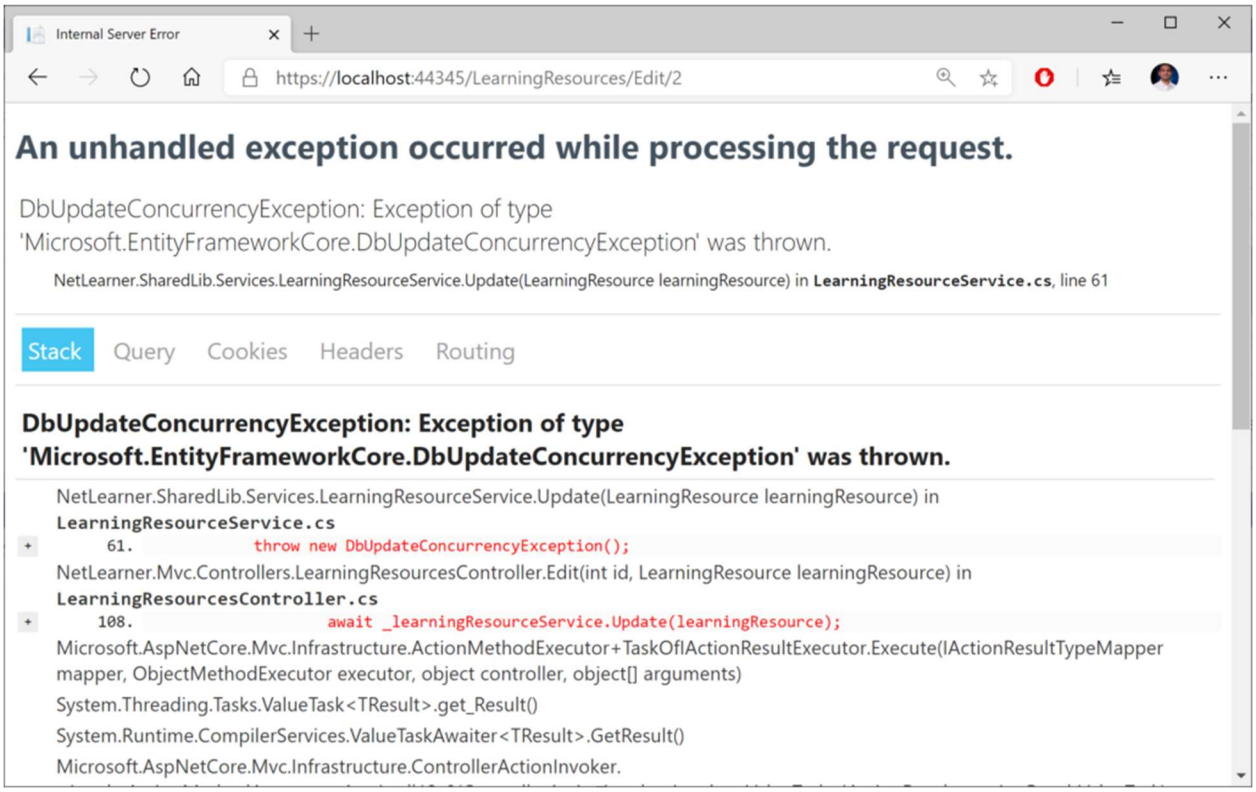
In the above code, you can also see a **throw** keyword by itself, when the expected exception occurs if the Id is found to exist. In this case, the throw statement (followed immediately by a semicolon) ensures that the exception is *rethrown*, while *preserving* the stack trace.

Run the MVC app and navigate to the Learning Resources page after adding some items. If there are no errors, you should see just the items retrieved from the database.



Learning Resources UI (no errors)

If an exception occurs in the Update() method, this will cause the expected exception to be thrown. To simulate this exception, you can intentionally throw the exception inside the update method. This should cause the error to be displayed in the web UI when attempting to save a learning resource.



Exception details in Web UI

Error Handling for MVC

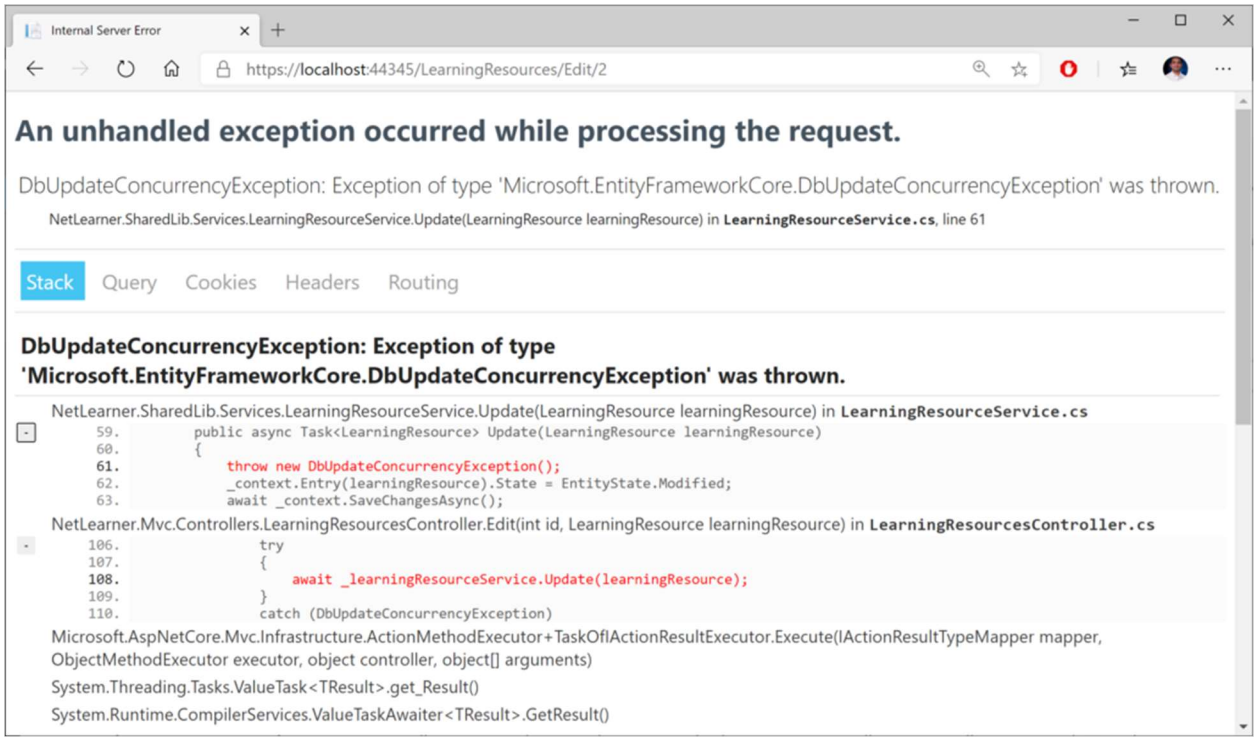
In ASP .NET Core MVC web apps, unhandled exceptions are typically handled in different ways, depending on whatever environment the app is running in. The default template uses the **DeveloperExceptionPage** middleware in a development environment but redirects to a shared Error view in non-development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

```
if (env.IsDevelopment())
{
   app.UseDeveloperExceptionPage();
   app.UseDatabaseErrorPage();
}
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.



Customized lines of code within error page

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 project'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
   });
}
```

But wait… what about **Web API** in ASP .NET Core? After posting the 2019 versions of 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();
   app.UseDatabaseErrorPage();
}
else
{
   app.UseExceptionHandler("/Error");
   ...
}
```

In the above code, you can see the that 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 for 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)

Error Handling for Blazor

In Blazor web apps, the UI for error handling is included in the Blazor project templates. Consequently, this UI is available in the NetLearner Blazor sample as well. The _Host.cshtml file in the Pages folder holds the following <**environment**> elements:

```
<div id="blazor-error-ui">
     <environment include="Staging,Production">
         An error has occurred. This application may no longer respond
until reloaded.
     </environment>
     <environment include="Development">
         An unhandled exception has occurred. See browser dev tools
for details.
     </environment>
     <a href="" class="reload">Reload</a>
     <a class="dismiss">✕</a>
 </div>
```

The div identified by the id "blazor-error-ui" ensures that is hidden by default, but only shown when an error has occured. Server-side Blazor maintains a connection to the end-user by preserving state with the use of a so-called circuit.

Each browser/tab instance initiates a new circuit. An unhandled exception may terminate a circuit. In this case, the user will have to reload their browser/tab to establish a new circuit.

According to the official documentation, unhandled exceptions may occur in the following areas:

1. **Component instantiation**: when constructor invoked
2. **Lifecycle methods**: (see Blazor post for details)
3. **Upon rendering**: during BuildRenderTree() in .razor component
4. **UI Events**: e.g. onclick events, data binding on UI elements
5. **During disposal**: while component's .Dispose() method is called
6. **JavaScript Interop**: during calls to IJSRuntime.InvokeAsync<T>
7. **Prerendering**: when using the Component tag helper

To avoid unhandled exceptions, use try-catch exception handlers within .razor components to display error messages that are visible to the user. For more details on various scenarios, check out the official documentation at:

- Handle errors in ASP.NET Core Blazor apps: https://docs.microsoft.com/en-us/aspnet/core/blazor/handle-errors?view=aspnetcore-3.1

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 of care default Logging setup and configuration (*behind the scenes*).

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

The Web Host Builder was replaced by the Generic Host Builder in ASP .NET Core 3.0, so it looks slightly different now. 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.

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
     Host.CreateDefaultBuilder(args)
         .ConfigureWebHostDefaults(webBuilder =>
         {
```

```
            webBuilder.UseStartup();
        });
```

The host can be used to set up logging configuration, e.g.:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.ClearProviders();
            logging.AddConsole(options => options.IncludeScopes =
true);
            logging.AddDebug();
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup();
        });
```

To make use of error logging (in addition to other types of logging) in your web app, you may call the necessary methods in your controller's action methods or equivalent. 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)

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
- Using Polly for .NET Resilience with .NET Core: https://www.telerik.com/blogs/using-polly-for-net-resilience-and-transient-fault-handling-with-net-core

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

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

References

- 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
- Logging: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging
- Handle errors in ASP.NET Core Blazor apps: https://docs.microsoft.com/en-us/aspnet/core/blazor/handle-errors?view=aspnetcore-3.1

IIS Hosting for ASP .NET Core 3.1 Web Apps

By Shahed C on March 2, 2020

4 Replies

This is the ninth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Handling Errors in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.9-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.9-alpha

In this Article:

I is for IIS Hosting

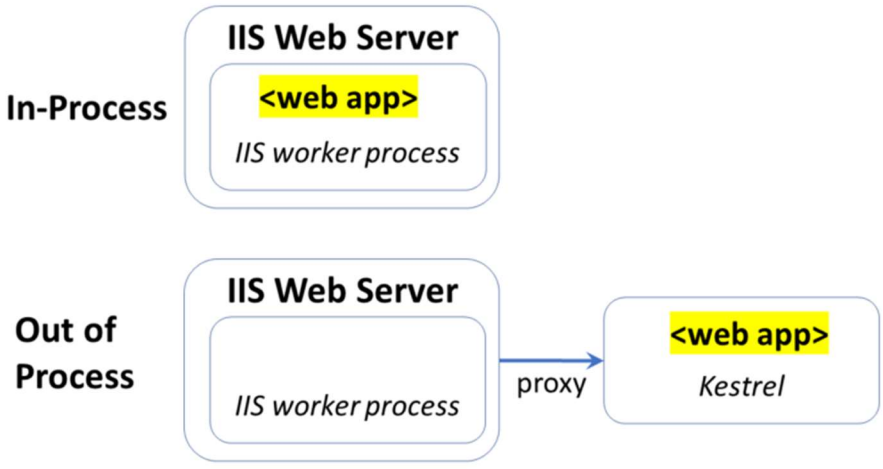
If you've been reading my weekly blog posts in this series (or you've worked with ASP .NET Core), you probably know that ASP .NET Core web apps can run on multiple platforms. Since Microsoft's IIS (Internet Information Services) web server only runs on Windows, you may be wondering why we would need 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, which continues to be available in v3.1. You can run your app either in-process or out of process.

Here's a little background, 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."

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



In-Process vs Out-of-Process

Developing for IIS

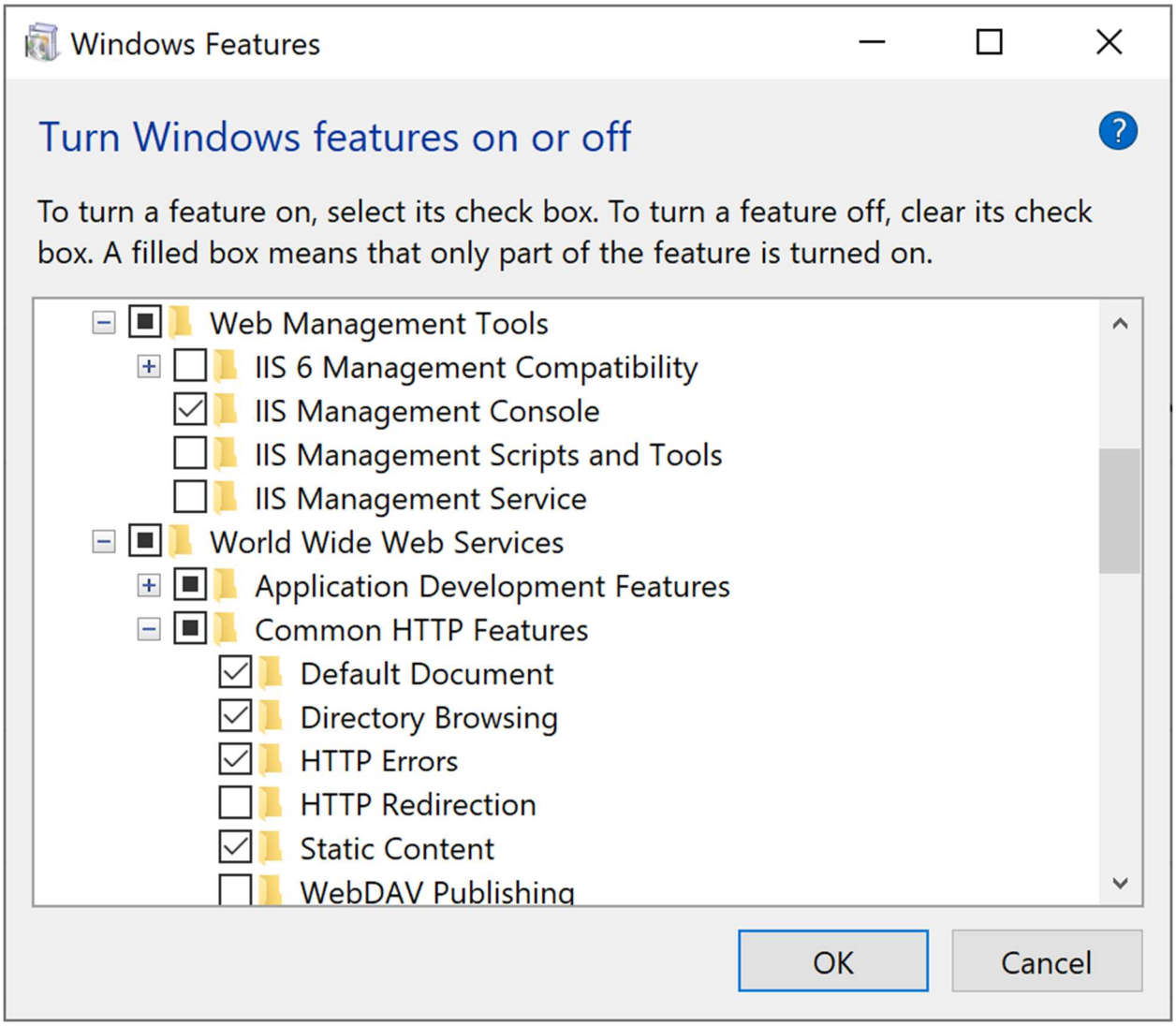
In order to develop for IIS on your local development machine while working with Visual Studio, you should complete the following steps:

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:

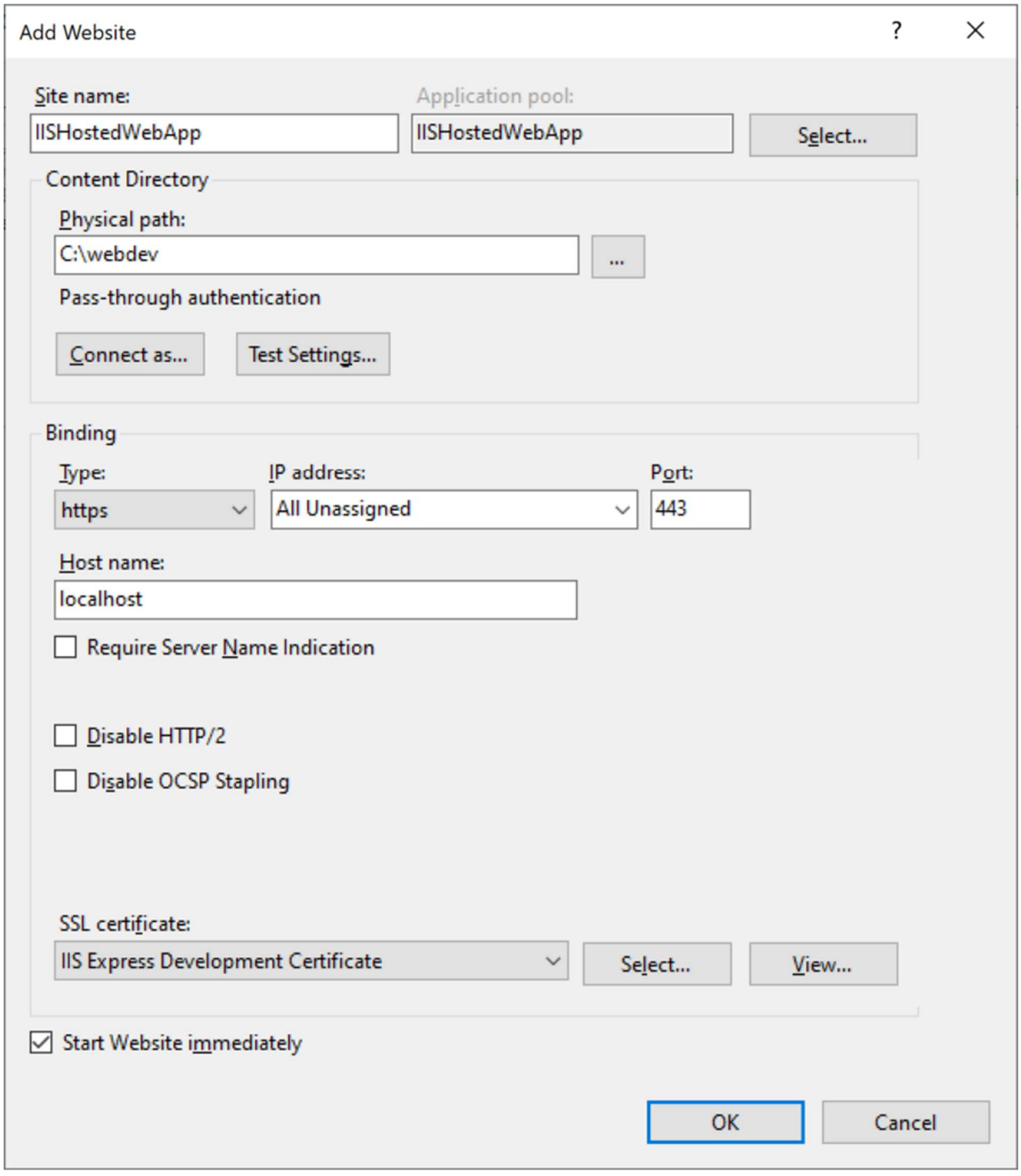
- Development-time IIS support in Visual Studio for ASP.NET Core: https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/development-time-iis-support?view=aspnetcore-3.1

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

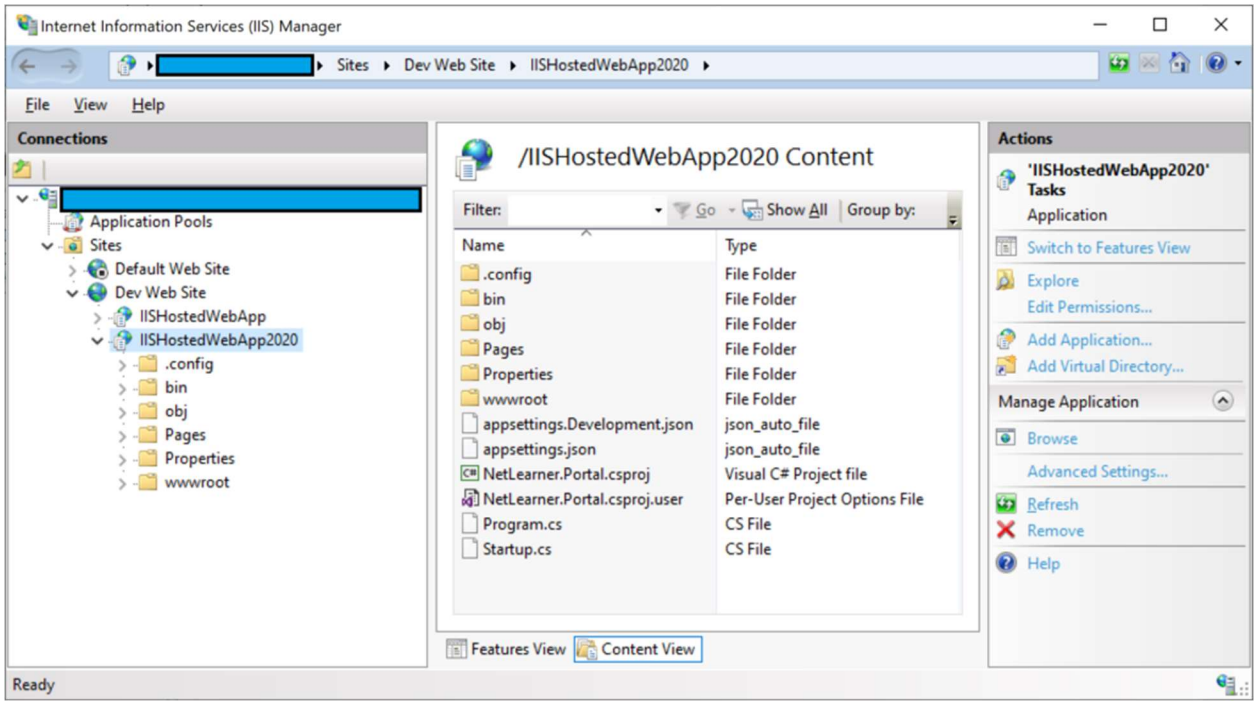


IIS components in Windows Features

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

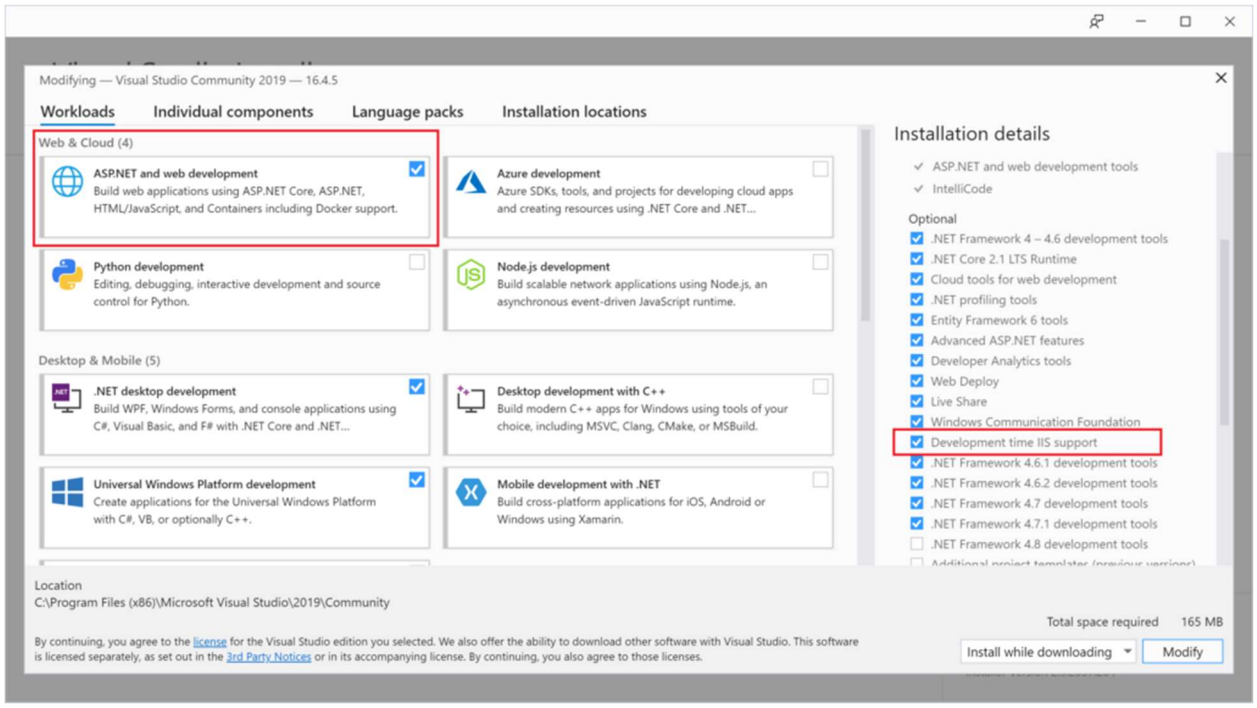


Add Website in IIS Manager



Browse Website Folder via IIS Manager

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:



Development time IIS Support in VS2019 ASP .NET workload

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. You may also need to start VS2019 with Admin privileges before attempting to debug against IIS.

Optionally, 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:

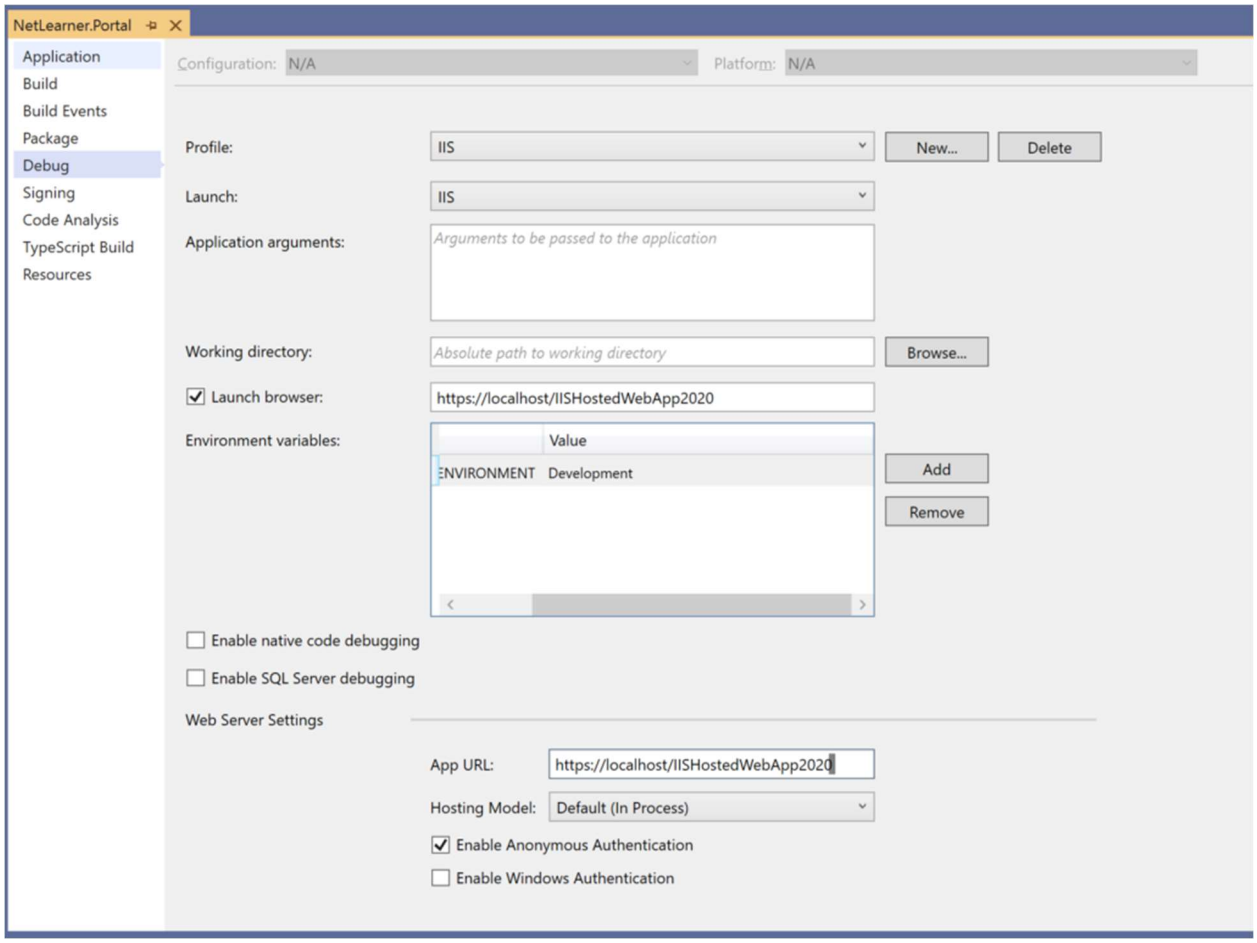
- Create Windows VM in Azure portal: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/quick-create-portal
- Create VMs running an IIS, etc: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/tutorial-iis-sql
- (Pre-configured) IIS on Windows Server 2016: https://azuremarketplace.microsoft.com/en-us/marketplace/apps/apps-4-rent.iis-on-windows-server-2016
- (Pre-configured) IIS on Windows Server 2019: https://azuremarketplace.microsoft.com/en-us/marketplace/apps/apps-4-rent.iis-on-windows-server-2019

IIS Configuration in Visual Studio

After completing Step 4 using the instructions outlined above, let's observe the following (in the NetLearner.Portal web app project:

- **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. Recall that the Hosting Model offers 3 options during the creation of this profile: Default, In-Process and Out-of-process.



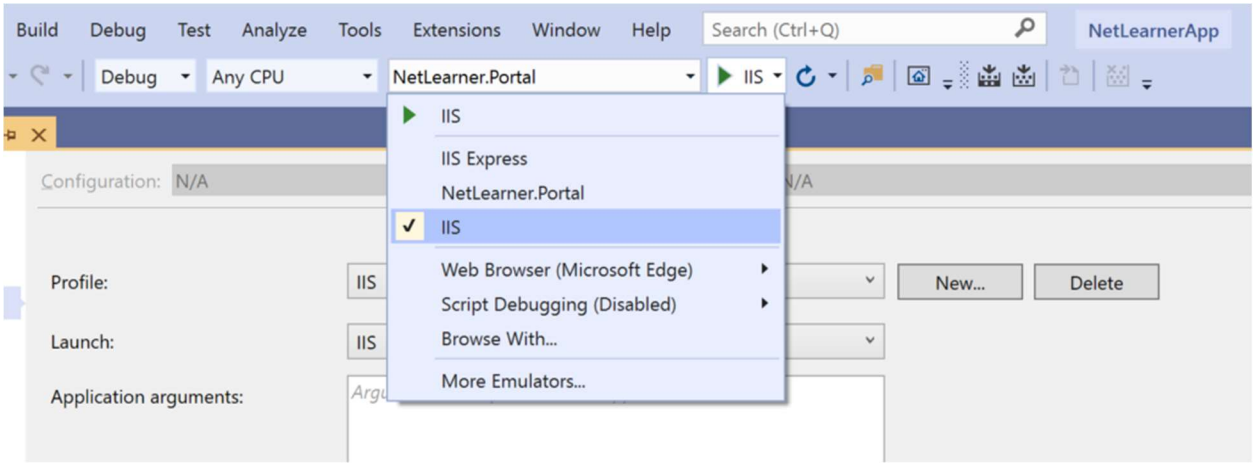
Settings: Under the Properties node in the Solution Explorer, open the ***launchSettings.json*** file. Here, you can see all the settings for your newly-created IIS-specific Profile. You can manually add/edit settings directly in this file without having to use the Visual Studio UI.

launchSettings.json file, showing IIS Profile settings

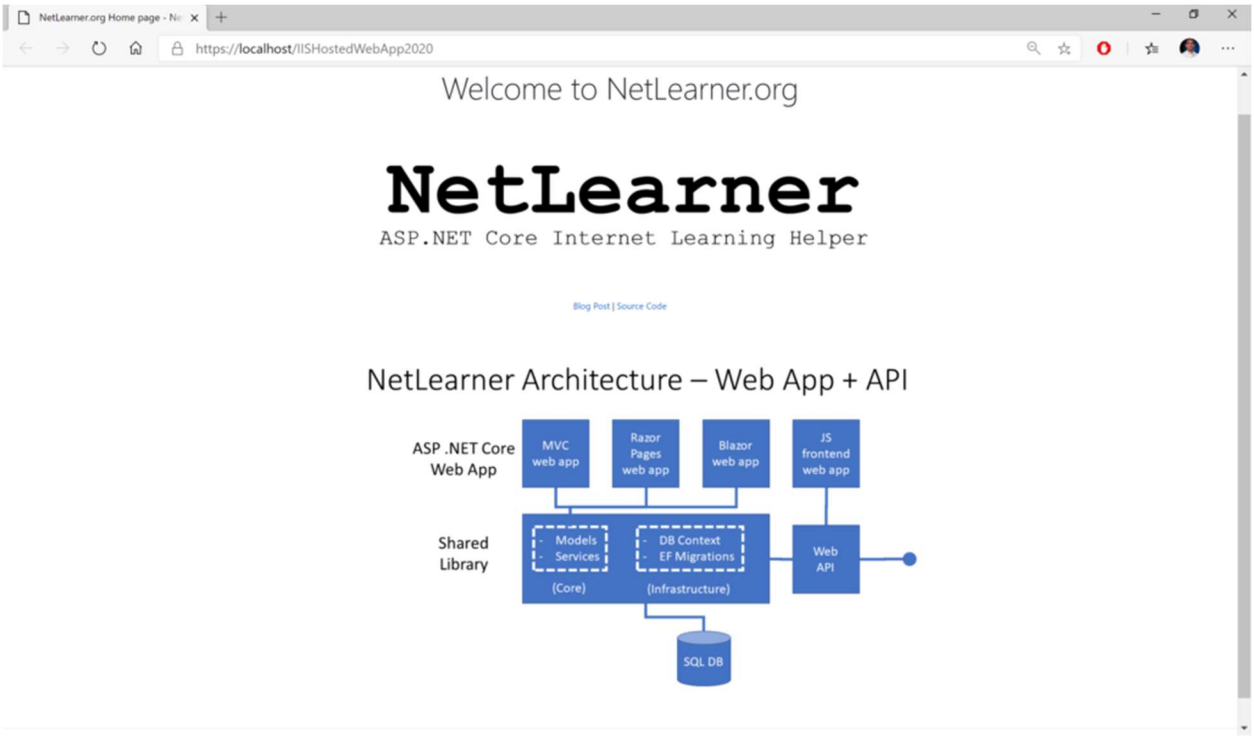
Config: You may recognize the XML-based web.config file from previous versions of ASP .NET, but the project's settings and dependencies are now spread across multiple files, such as the .csproj file and appsettings.json config file. For ASP .NET Core projects, the web.config file is only used specifically 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.

web.config file, showing hosting model, e.g. InProcess

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>



VS2019 with IIS Profile selected



Web browser showing web app running in IIS

ASP .NET Core Module

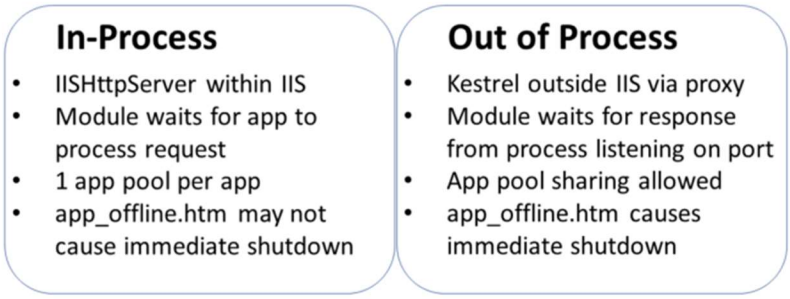
Now that your IIS server environment has been set up (either locally or in the cloud), let's learn about the ASP .NET Core Module. This is a native IIS module that plugs into the actual IIS pipeline (not to be confused with your web app's request pipeline). It allows you to host your web app in one of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

```
 <aspNetCore
processPath="C:\Users\REPLACE_SOMEUSER\source\repos\NetLearnerApp\src\
NetLearner.Portal\bin\Debug\netcoreapp3.1\NetLearner.Portal.exe"
arguments="" stdoutLogEnabled="false" hostingModel="InProcess">
```

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



In-Process vs Out-of-Process. compared

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 (first introduced) 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 3.1

By Shahed C on March 9, 2020

4 Replies

This is the tenth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- IIS Hosting for ASP .NET Core 3.1 Web Apps

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.10-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.10-alpha

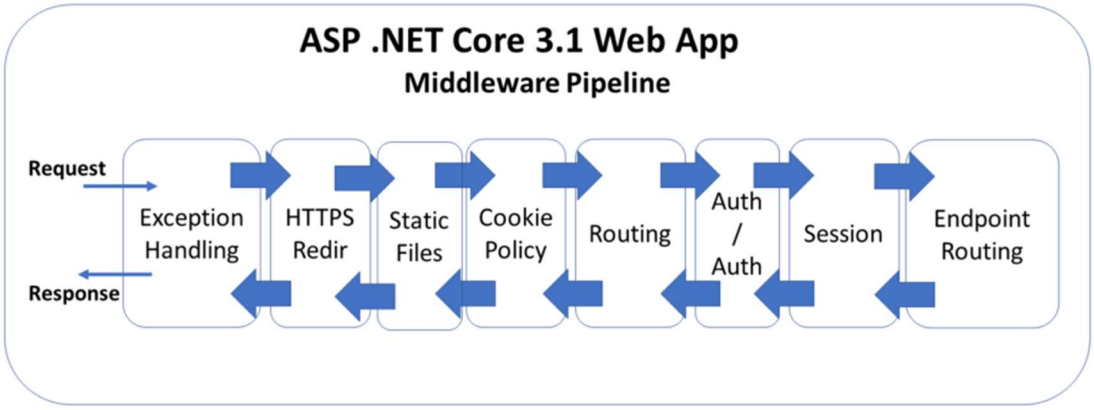
In this Article:

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, Razor Pages or Blazor), 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.



ASP .NET Core 3.1 middleware pipeline diagram

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

- MVC – https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Mvc
- Razor Pages – https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Pages

- Blazor – https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Blazor

Configuring Static Files via Middleware

Let's start by observing the Startup.cs configuration file, from any of the web projects. 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();
    ...
}
```

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, Razor Pages or Blazor project.

It's useful to note the placement of this line of code. It appears *before* app.**UseRouting**() and app.**UseEndpoints**(), 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 application routing middleware. This becomes even more important when authentication is used.

Take a look at any of the web projects to observe the 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(...);
}
```

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, routing, etc) 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 IHostBuilder CreateHostBuilder(string[] args) =>
      Host.CreateDefaultBuilder(args)
         .ConfigureWebHostDefaults(webBuilder =>
         {
            webBuilder.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.

```
Host.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.

```
webBuilder.UseWebRoot("<content-root>")
```

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.

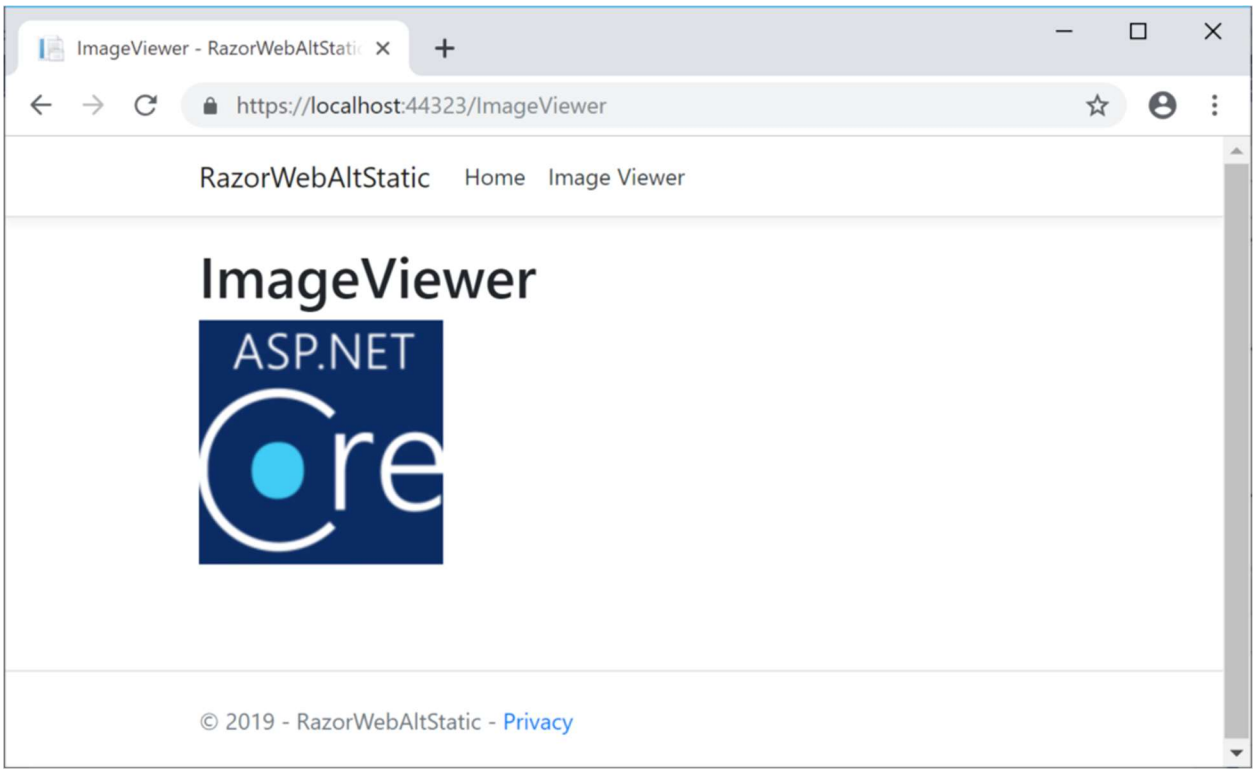
The following markup may be used in a .cshtml file to refer to an image, e.g. MyImage01.png:

```
<img src="~/AltStaticFiles/MyImages/MyImage01.png" />
```

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

Visual Studio showing image loaded from alternate location

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



Web Browser showing image loaded from alternate location

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 Identity pages, e.g. _ValidationScriptsPartial.cshtml in the Razor sample code.

```
<environment include="Development">
  <script src="~/Identity/lib/jquery-
validation/dist/jquery.validate.js"></script>
</environment>

<environment exclude="Development">
 <script
src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.17.0/jquery.val
idate.min.js"
asp-fallback-src="~/Identity/lib/jquery-
validation/dist/jquery.validate.min.js"
asp-fallback-test="window.jQuery && window.jQuery.validator"
crossorigin="anonymous"
integrity="sha384-
rZfj/ogBloos6wzLGpPkkOr/gpkBNLZ6b6yLy4o+ok+t/SAKlL5mvXLr0OXNi1Hp">
    </script>
</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 jQuery Validate file is obtained from your local copy. When *not* in development (e.g. staging, production, etc), the jQuery Validate file is obtained from a CDN.

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 newer LibMan (aka Library Manager) for easily adding and updating your client-side libraries.

LibMan (aka Library Manager)

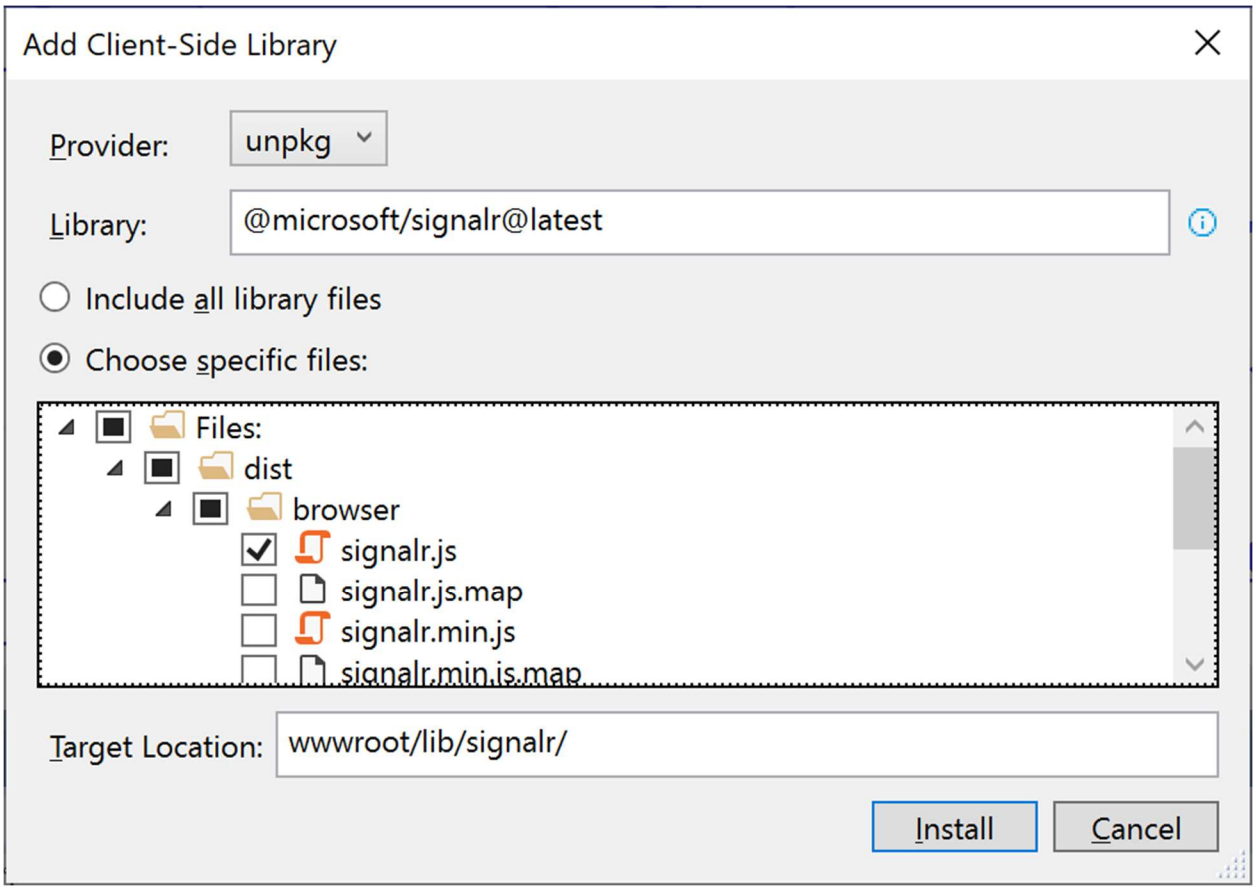
The easiest way to use LibMan is to use the built-in features available in Visual Studio. Using LibMan using the IDE is as easy as launching it from Solution Explorer. Specify the provider from the library you want, and any specific files you want from that library.

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. @microsoft/signalr@latest to pick the latest stable version of the SignalR client-side library
- **Files**: At a minimum, choose specific files, e.g. signalr.js and/or its minified equivalent
- **Target Location**: modify the target location folder tree as desired



Adding a client-side library

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 or the newer NetLearner.Portal app's libman.json file to observe its syntax for using a SignalR client library.

```
{
  "version": "1.0",
  "defaultProvider": "unpkg",
  "libraries": [
    {
      "library": "@microsoft/signalr@latest",
      "destination": "wwwroot/lib/signalr/",
      "files": [
        "dist/browser/signalr.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 a 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/

Key Vault for ASP .NET Core 3.1 Web Apps

By Shahed C on March 16, 2020

5 Replies

This is the eleventh of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- JavaScript, CSS, HTML & Other Static Files in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.11-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.11-alpha

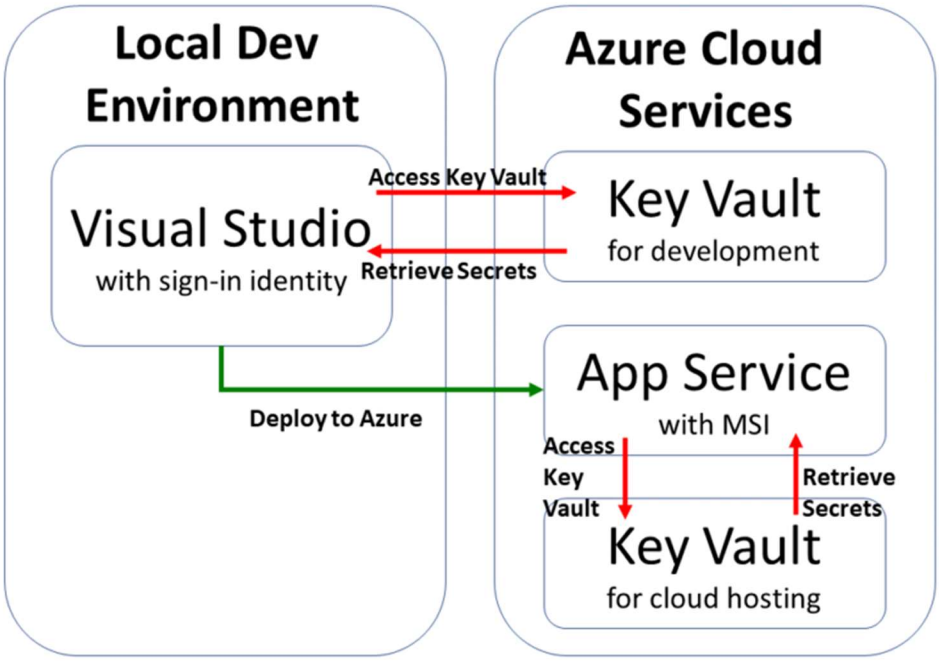
In this Article:

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



Using Key Vault from Azure for your Web Apps

Setting up Key Vault in Azure

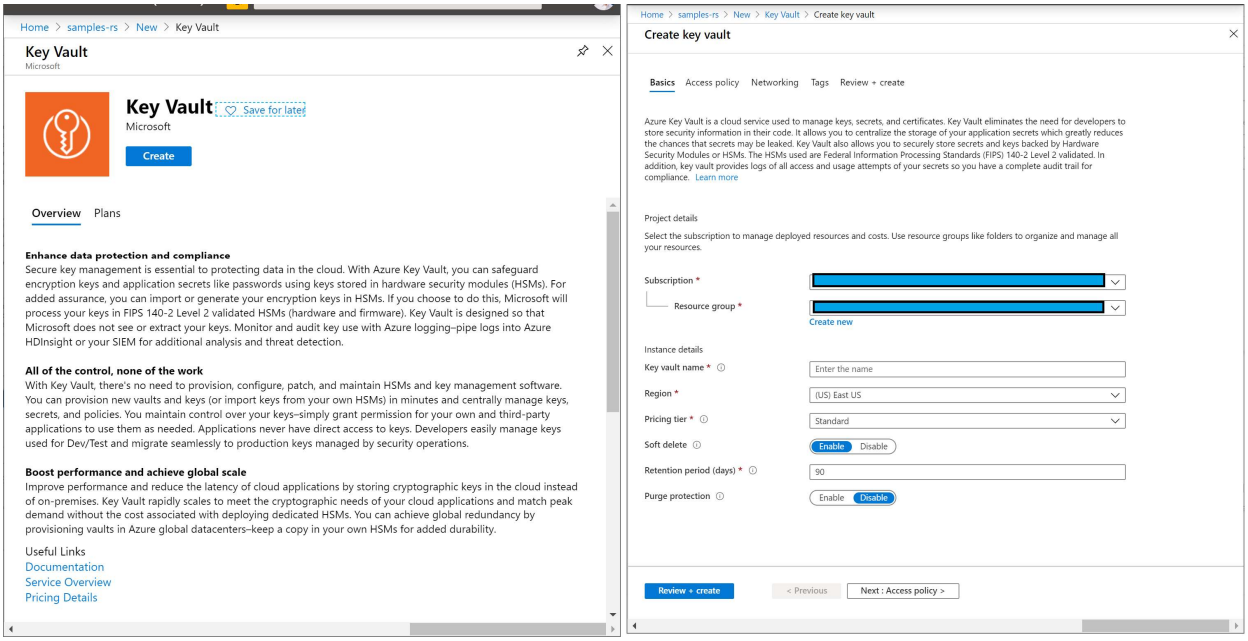
Before you can use Key Vault in the cloud, you will have to set it up in Azure for your use. This can be done in various ways:

- **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:

- *Subscription:* select the desired Azure subscription
- *Resource Group*: select a resource group or create a new one

- *Name:* alphanumeric, dashes allowed, cannot start with number
- *Region:* select the desired region
- *Pricing Tier*: select the appropriate pricing tier (Standard, Premium)
- *Soft Delete*: enable/disable
- *Retention Period:* enter retention period (in days)
- *Purge protection:* enable/disable



Key Vault creation screen on Azure Portal

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.

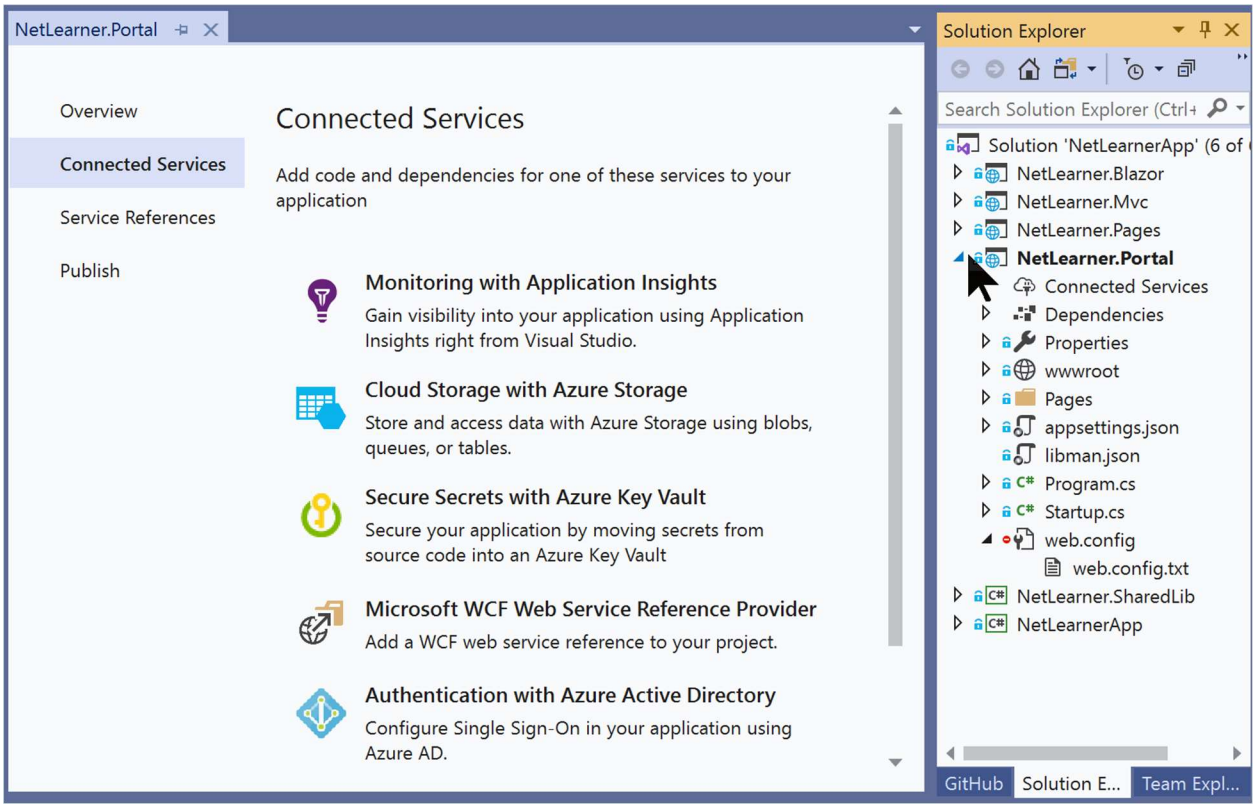
```
az keyvault secret set --vault-name "MyKeyVault" --name "MySecretName" --value "MySecretValue"
```

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.

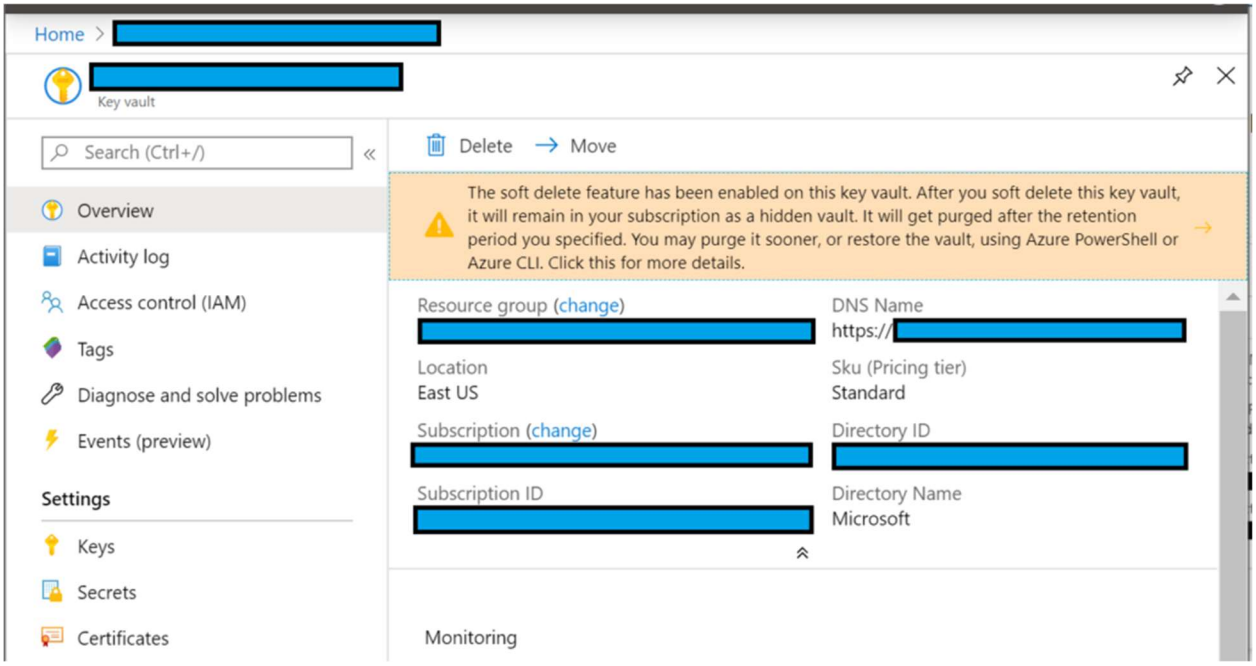
If you need help with Visual Studio, check out the official docs at:



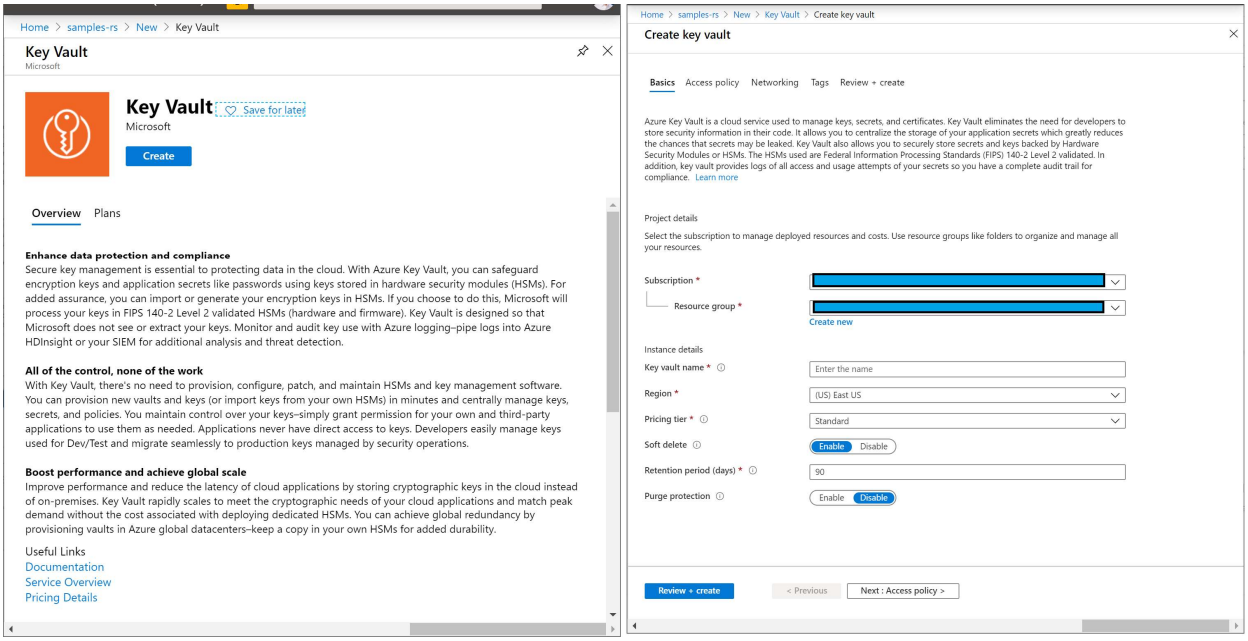
Add a Connected Service in Visual Studio

- 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

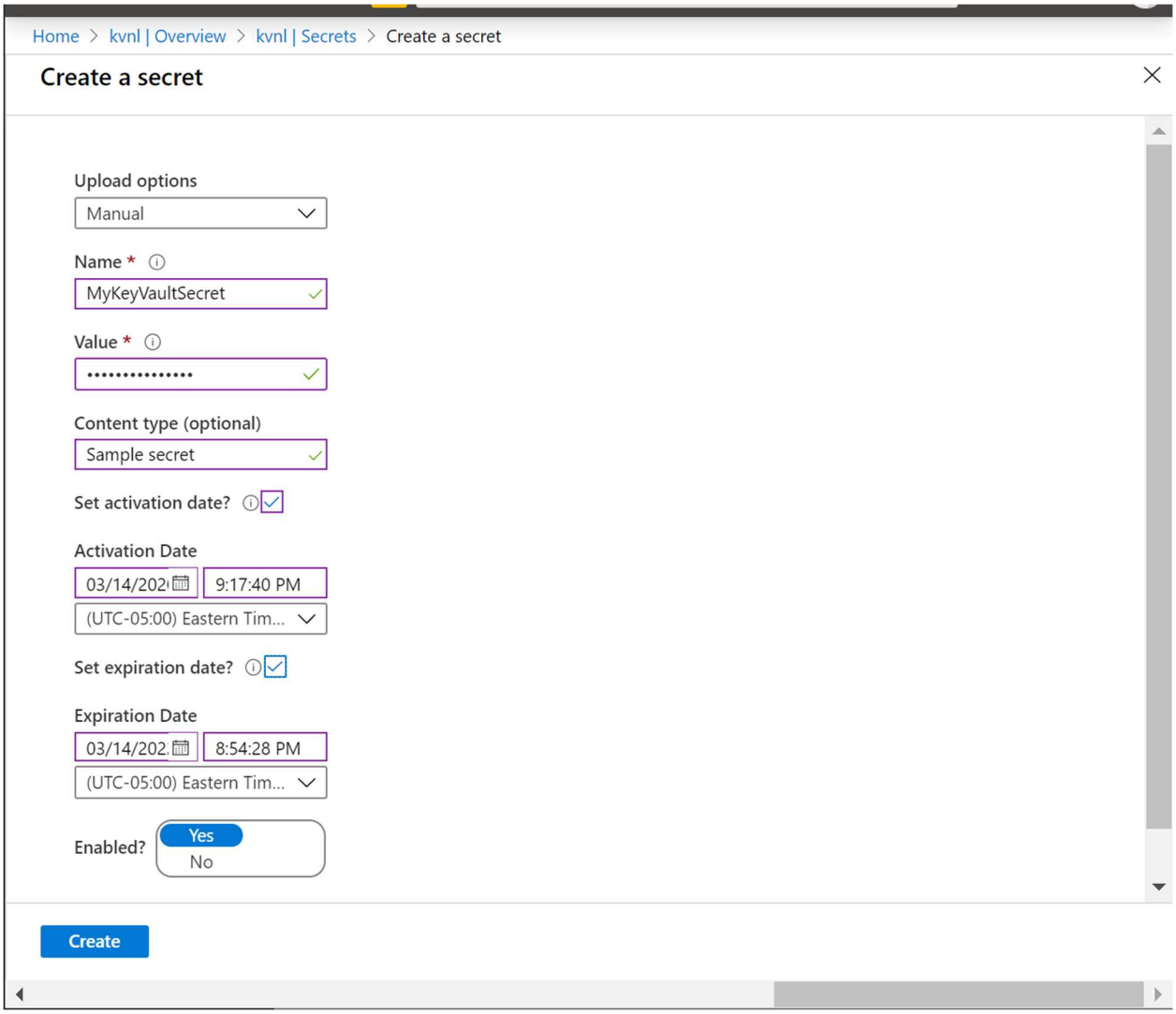
Once created, the Key Vault and its secret names (and values) can be browsed in the Azure Portal, as seen in the screenshots below:



Key Vault details



Key Vault creation in Azure Portal



Creating a secret for Azure 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, KeyVault and AzureKeyVault NuGet libraries.
- Microsoft.Azure.Services.AppAuthentication
- Microsoft.Azure.KeyVault
- Microsoft.Extensions.Configuration.AzureKeyVault
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 IHostBuilder CreateHostBuilder(string[] args) =>
   Host.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());
          }
       }
   ).ConfigureWebHostDefaults(webBuilder =>
   {
      webBuilder.UseStartup<Startup>();
   });

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

NOTE: *Please note that usage of Web Host Builder in ASP .NET Core 2.x has been replaced by the Generic Host Builder in .NET Core 3.x.*

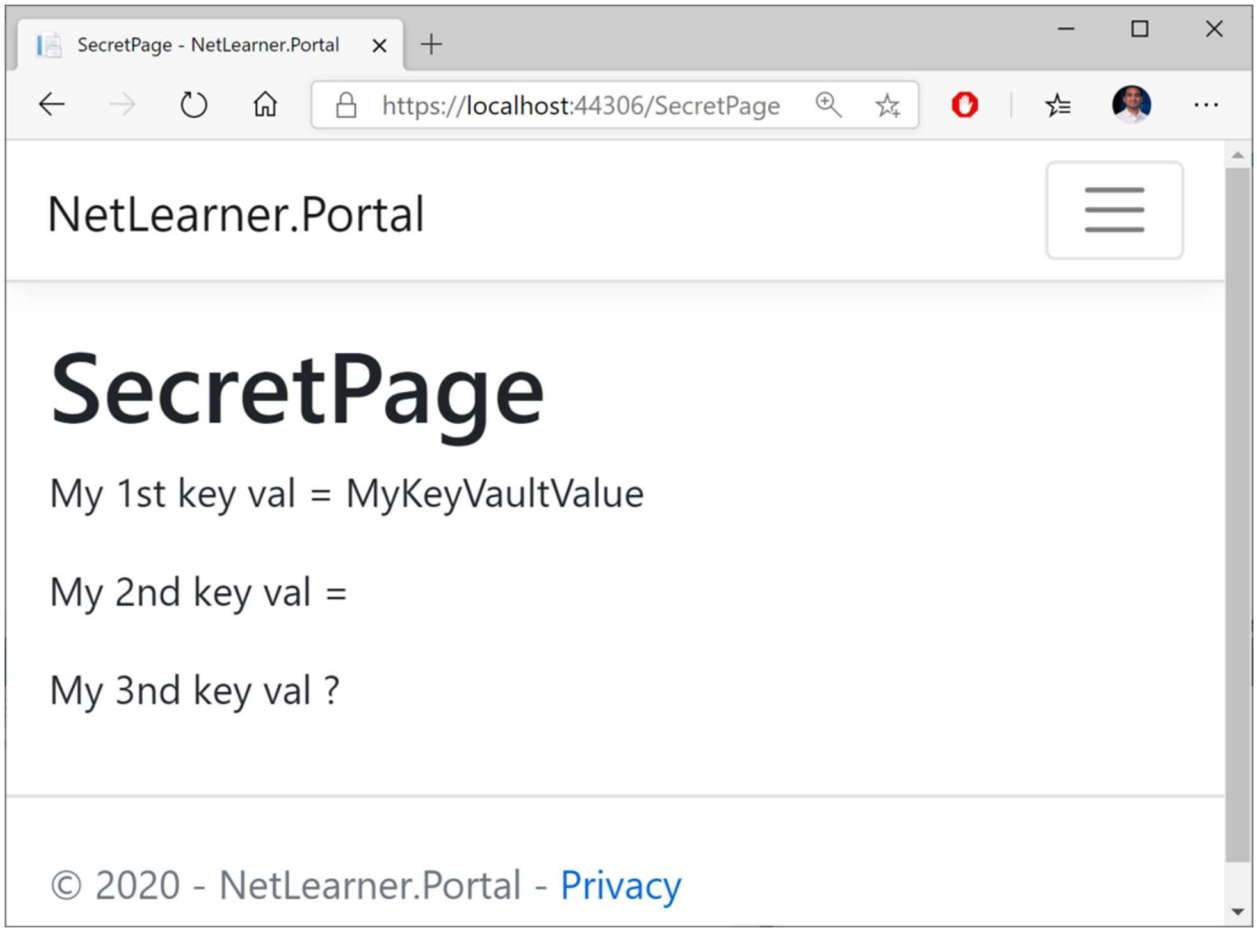
Below is an example of of a Page Model for a Razor Page, e.g. SecretPage.cshtml.cs in the sample app

```
 ...
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*).



SecretPage showing secrets retrieved from Key Vault

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:

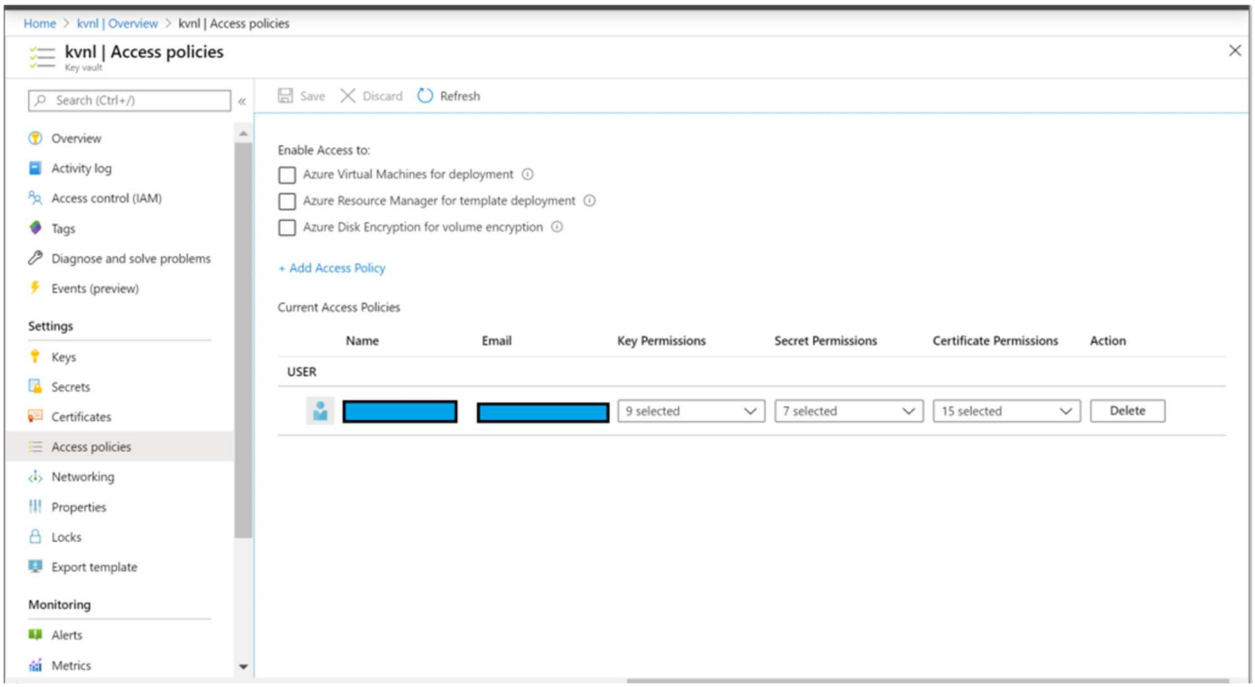
- Deploying ASP .NET Core to Azure App Service: https://wakeupandcode.com/deploying-asp-net-core-to-azure-app-service/

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

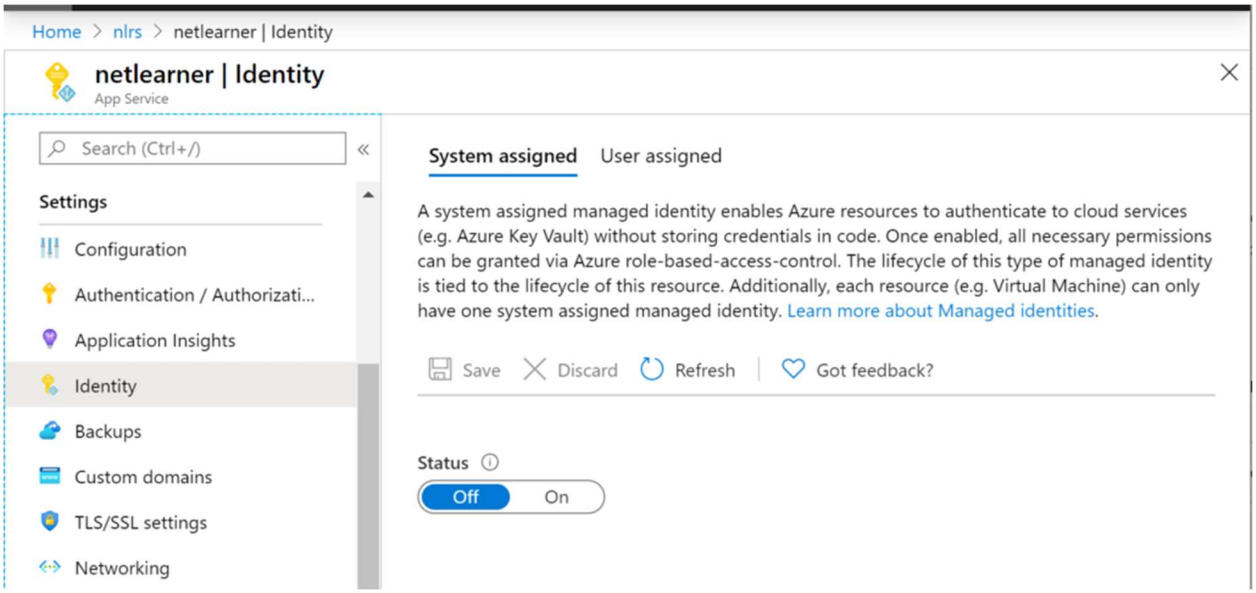
- **Azure Portal**: log in to the Azure Portal and add your app
- **Azure CLI**: use Azure CLI commands to set up MSI
- **Visual Studio**: use the VS IDE while publishing

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*"



Key Vault Access Policies



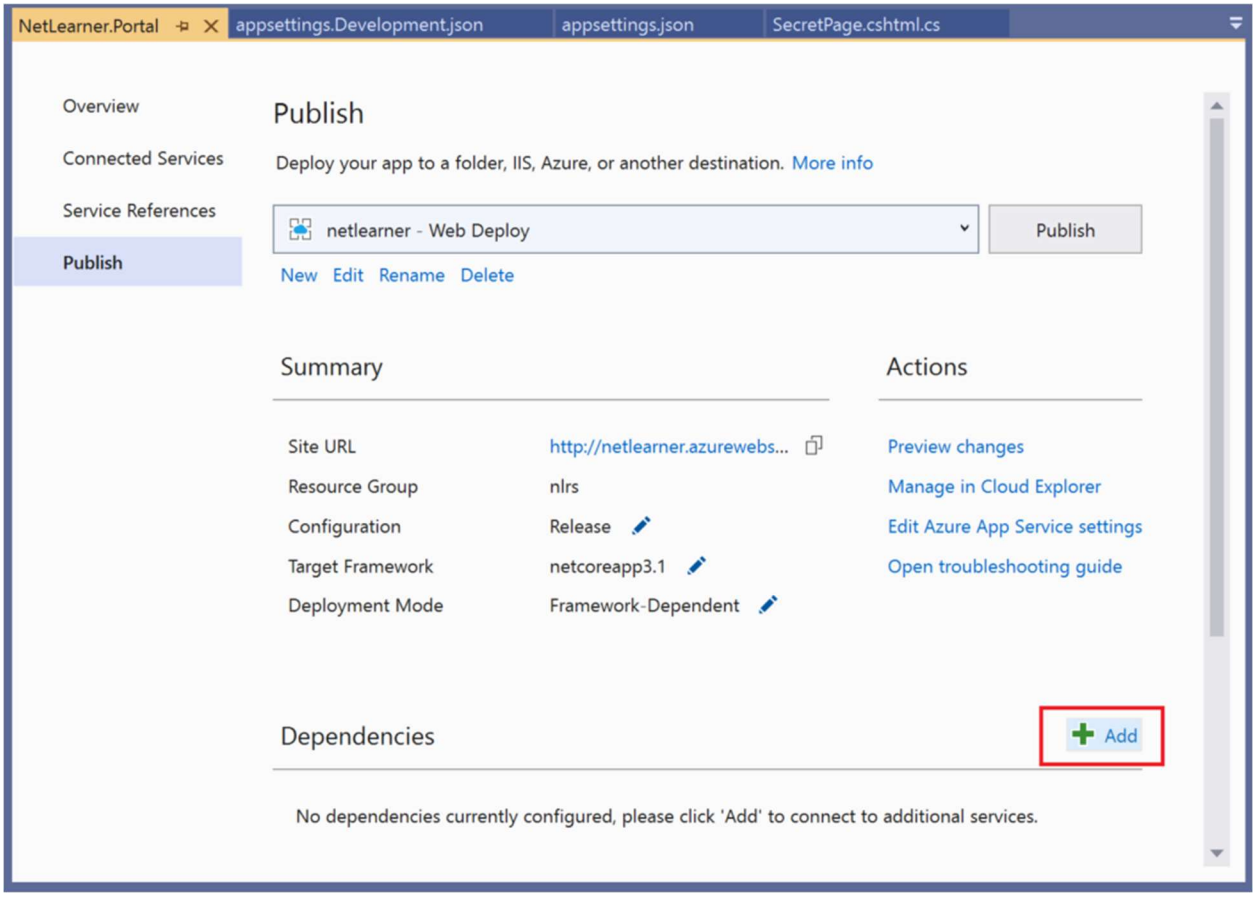
Key Vault App Service Identity

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.

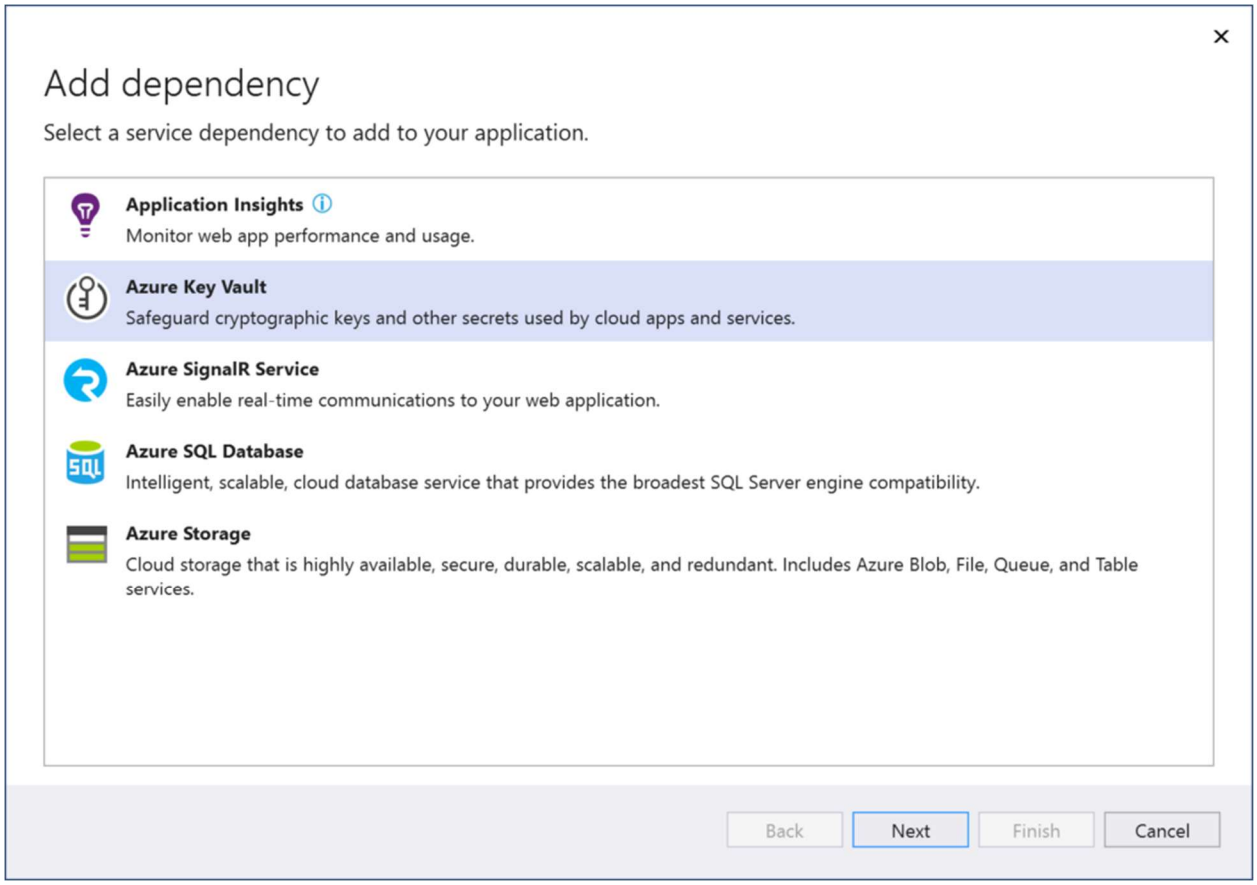
```
az keyvault set-policy --name "MyKeyVault" --spn <APP_ID> --secret-permissions get
```

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 Dependencies, including Azure

Key Vault. After you've added and enabled Key Vault for your application, the option will change to say "Configure" and "Manage Key Vault".



Add Dependencies during Publish from Visual Studio



Select Azure Key Vault from list of dependencies

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/dotnet/AspNetCore.Docs/tree/master/aspnetcore/security/key-vault-configuration/samples/3.x/SampleApp

- Getting Started with Azure Key Vault with .NET Core: https://azure.microsoft.com/en-us/resources/samples/key-vault-dotnet-core-quickstart/
- 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/slide-decks/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 3.1

By Shahed C on March 24, 2020

5 Replies

This is the twelfth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Key Vault for ASP .NET Core 3.1 Web Apps

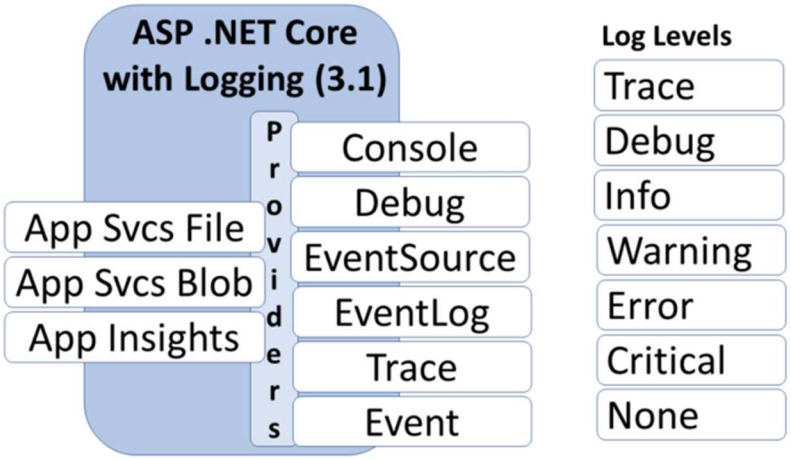
NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.12-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.12-alpha

In this Article:

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, you should definitely consider a robust semantic/structured logging solution for your projects.



Logging providers in ASP .NET Core 3.1

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 NetLearner.Portal 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 adds the Console and Debug providers. As of ASP.NET Core 2.2, the EventSource provider is also automatically added by the default builder. When on Windows, the EventLog provider is also included.

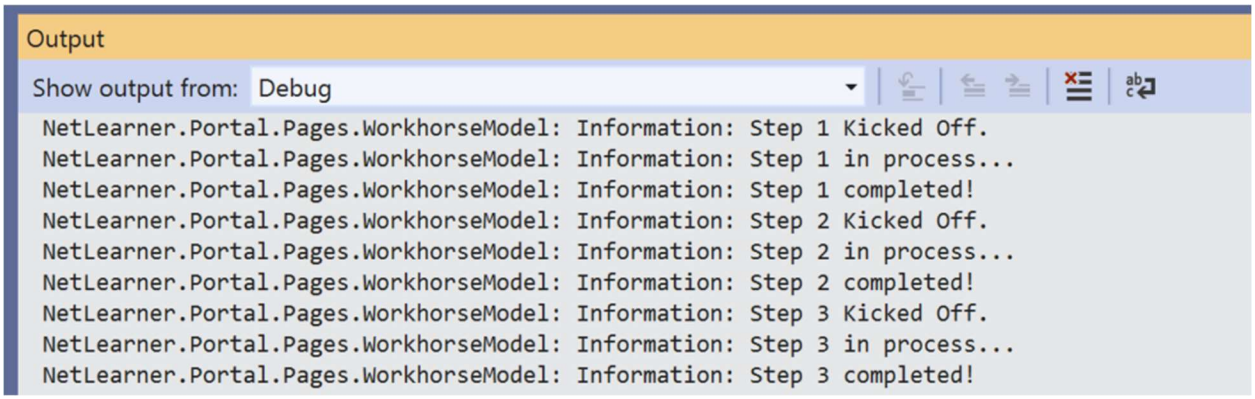
```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
         webBuilder.UseStartup<Startup>();
      });
```

NOTE: As mentioned in an earlier post in this blog series, the now-deprecated Web Host Builder has been replaced by the Generic Host Builder with the release of .NET Core 3.0.

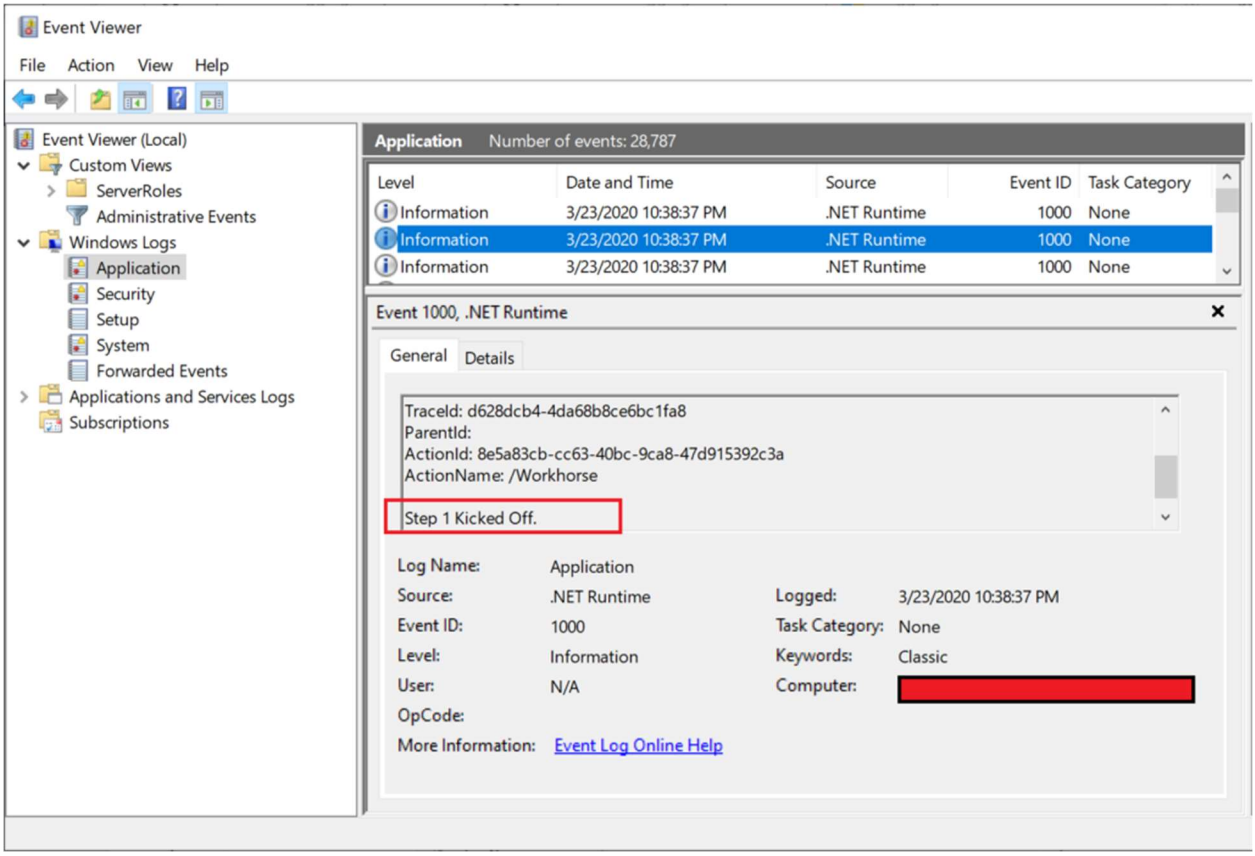
If you wish to add your own set of logging providers, you can expand the call to CreateDefaultBuilder(), clear the default providers, and then add your own. The built-in providers now include **Console**, **Debug, EventLog, TraceSource** and **EventSource.**

```
public static IHostBuilder CreateHostBuilder(
   string[] args) => Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
         webBuilder.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.



VS2019 Output panel showing debug messages



Windows Event Viewer showing log data

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?view=netcore-3.1

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

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-3-1-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.

```
  .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

To set the EventLog level explicitly, add a section for “EventLog” with the default minimum “LogLevel” of your choice. This is useful for Windows Event Logs, because the Windows Event Logs are logged for Warning level or higher, by default.

```
{
   "Logging": {
     "LogLevel": {
       "Default": "Debug"
     },
     "EventLog": {
```

```
      "LogLevel": {
        "Default": "Information"
      }
    }
  }
}
```

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 WorkhorseModel class seen in The NetLearner.Portal project, the log results seen in the Debug window started with:

```
NetLearner.Portal.Pages.WorkhorseModel
```

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

```
private readonly ILogger _logger;

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

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

```
private readonly ILogger _logger;

public WorkhorseModel(WorkhorseModel logger)
{
   _logger =
logger.CreateLogger("NetLearner.Portal.Pages.WorkhorseModel");
}
```

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

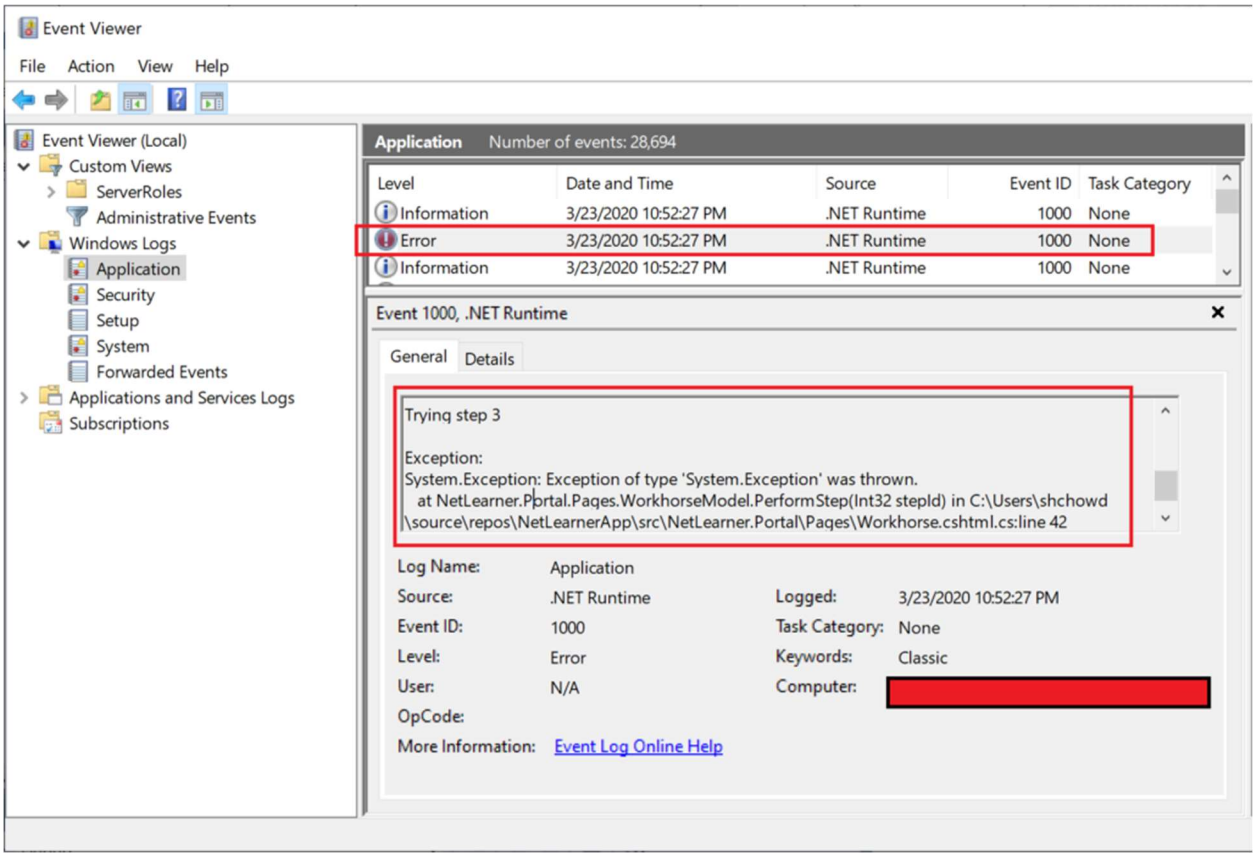
1. Instead of **ILogger**<*classname*> we are now passing in an **ILoggerFactory** type as the logger.
2. Instead of just assigning the injected logger to the private **_logger** variable, we are now calling the factory method **CreateLogger()** with the desired string value to set the category name.

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.



Windows Event Viewer showing error log entry

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
- Setting up Serilog in ASP.NET Core 3: https://nblumhardt.com/2019/10/serilog-in-aspnetcore-3/

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/
- Explore .NET trace logs in Azure Application Insights with ILogger: https://docs.microsoft.com/en-us/azure/azure-monitor/app/ilogger
- 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 3.1

By Shahed C on March 30, 2020

5 Replies

This is the thirteenth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Logging in ASP .NET Core 3.1

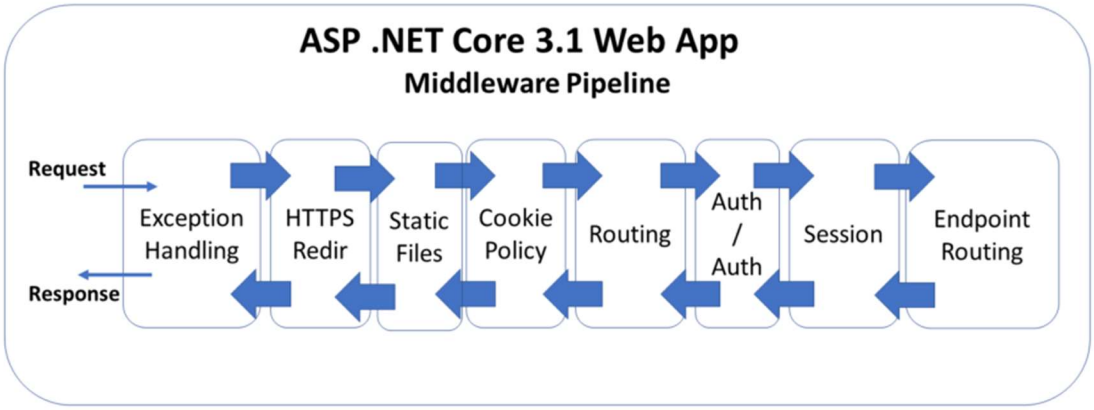
NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.13-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.13-alpha

In this Article:

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.



ASP .NET Core 3.1 Middleware Pipeline

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.

- Exception Handling
- HTTPS Redirection
- Static Files
- Cookie Policy
- Routing
- Authentication
- Authorization

- Session
- Endpoint Routing

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 use very useful across multiple scenarios, e.g. serving static files without the need for authentication, handling exceptions before anything else, etc.

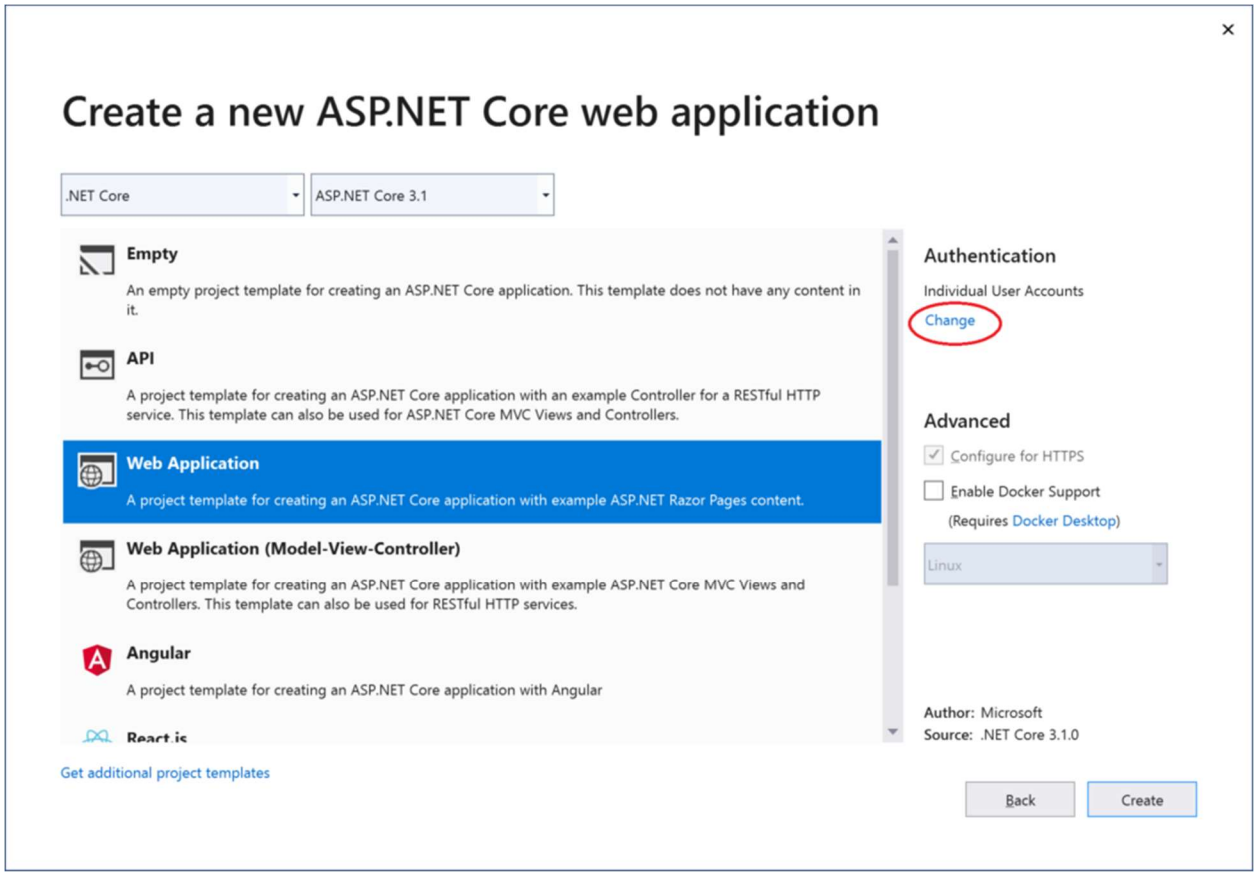
The returned response travels back in the reverse direction back through the pipeline. This allows each component to run code both times: when the request arrives and also when the response is on its way out.

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

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

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy(); // manually added

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseSession(); // manually added

    app.UseEndpoints(endpoints =>
```

```
    {
        // map MVC routes, Razor Pages, Blazor Hub
    });
}
```

You may add calls to **UseCookiePolicy()** and **UseSession**() manually, as these aren't included in the template-generated project. The endpoint configuration will vary depending on the type of project template you start with: MVC, Razor Pages or Blazor, which you can mix and match. You may also enable authentication and HTTPS when creating the template.



Adding a new project in VS2019

In order to configure the use of the Session middleware, you may add the following code in your **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;
```

```
    });
...
}
```

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.

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 mind when developing custom middleware.

***Exception Handling*:**

- **UseDeveloperExceptionPage**() & **UseDatabaseErrorPage**(): 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 endpoint routing middleware) ensures that cookies can be issued as necessary and that the user can be authenticated before the endpoint routing kicks in. For more information, check out the detailed post on Authentication & Authorization in ASP .NET Core.

Endpoint Routing:

- **UseEndpoints**(): usage varies based on project templates used for MVC, Razor Pages and Blazor.

- endpoints.**MapControllerRoute**(): set the default route and any custom routes when using MVC.
- endpoints.**MapRazorPages**(): sets up default Razor Pages routing behavior
- endpoints.**BlazorHub**(): sets up Blazor Hub

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.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
{
    ...
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
     });
}
```

You can use an alternative approach to terminate the pipeline, by calling **app.Use**() and then triggering the next middleware component with a call to **next(context)** as shown below. If there is no call to **next**(), then it essentially short-circuits the middleware pipeline.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
{
    // ...
    // Approach 1: terminal middleware
    app.Use(next => async context =>
    {
        if (context.Request.Path == "/")
        {
            await context.Response.WriteAsync("Hello and goodbye!");
            return;
        }
        await next(context);
    });
    app.UseRouting();
    app.UseEndpoints(...); // Approach 2: routing
}
```

Finally, there is the **Map**() method, which creates separate forked paths/branches for your middleware pipeline and multiple terminating ends. Check out the official documentation on mapped branches for more information:

- Branch middleware pipeline: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1#branch-the-middleware-pipeline

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.

In the official sample:

- accessing /*map1* in an HTTP request will call **HandleMapTest1**()
- accessing /*map2* in an HTTP request will call **HandleMapTest2**()
- 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

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

.NET 5.0, VS2019 Preview and C# 9.0 for ASP .NET Core developers

By Shahed C on April 6, 2020

3 Replies

This is the fourteenth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Middleware in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.14-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.14-alpha

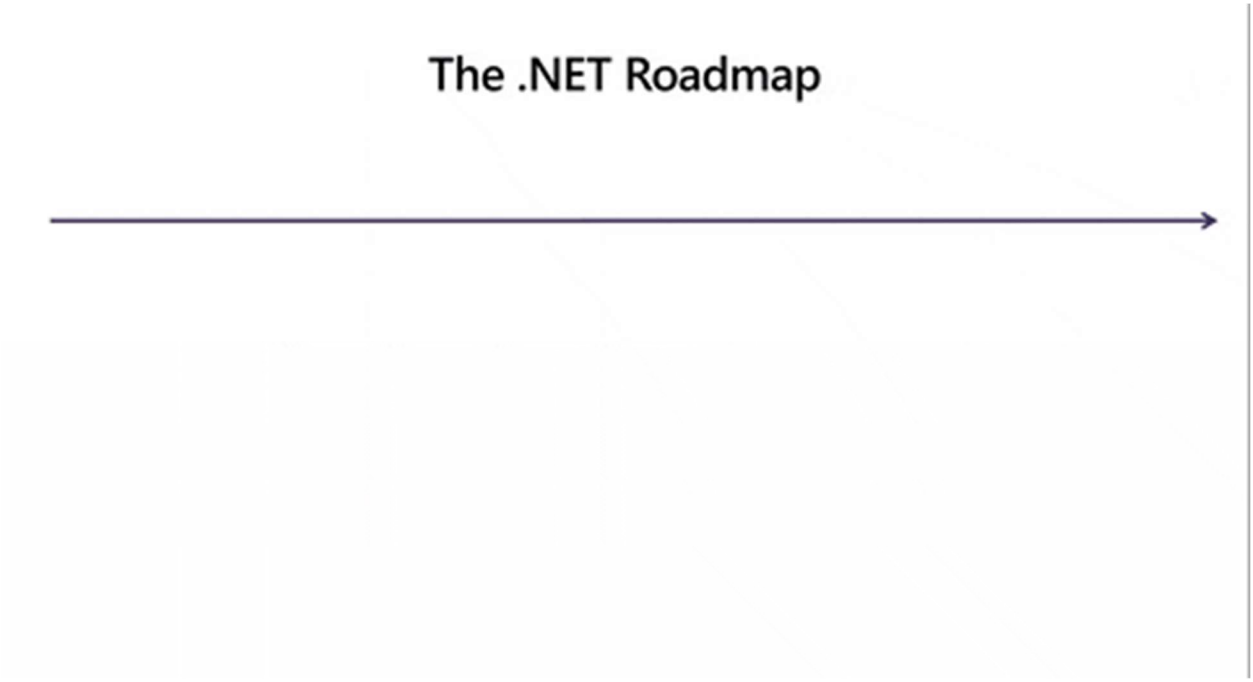
NOTE: The NetLearner suite of apps won't be updated to .NET 5.0 at this time, so you can check out the new template-generated projects in an experimental subfolder:

- Experimental projects with .NET 5: https://github.com/shahedc/NetLearnerApp/tree/main/experimental/NetLearner.DotNet5

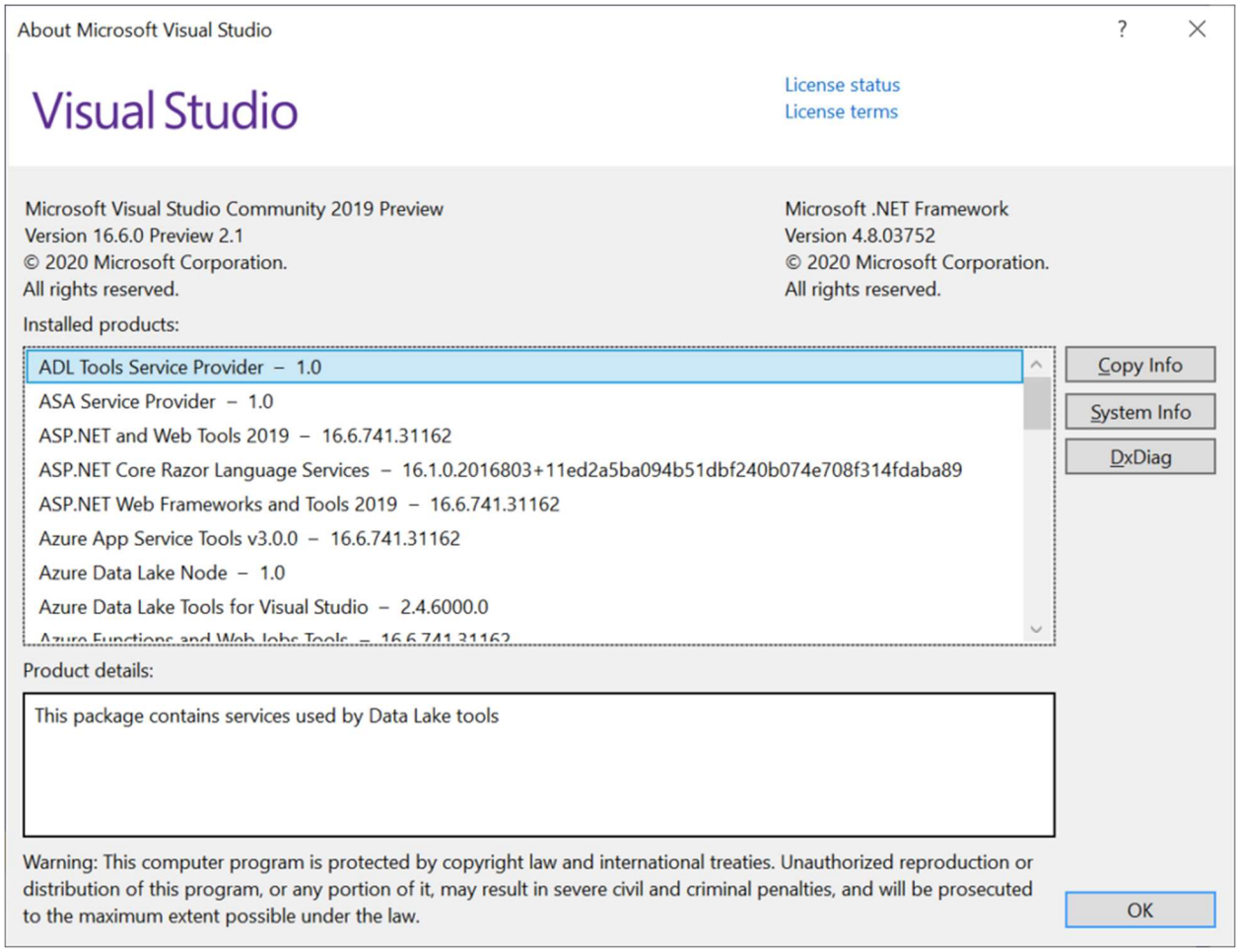
In this Article:

N is for .NET 5.0, VS2019 Preview and C# 9.0

As we have crossed the halfway point in this series, now is a good time to take a step back and look forward at .NET 5.0. This is the next iteration of .NET Core, which skips over the number 4, and unifies all flavors of .NET into a single .NET (i.e. .NET Core, .NET Framework and Mono).



In order to use .NET 5.0 (Preview 2 as of this writing), we need to install Visual Studio 2019 v16.6.0 Preview 2.1. This will give you access to all the latest (preview) project templates for ASP .NET Core 5.0. Since C# 9.0 is still early in development, we will just briefly touch on upcoming/proposed language features.



Visual Studio 2019 Preview

Visual Studio 2019 has been out since early 2019, so you can grab the latest stable version from the main download page. Visit any of the URLs below to select the edition you need:

- VS2019 Downloads: https://visualstudio.microsoft.com/downloads/
- Community Edition
- Professional Edition
- Enterprise Edition

In order to get the latest *Preview* version, you must visit the special download page for Previews:

- VS2019 Preview Downloads: https://visualstudio.microsoft.com/vs/preview/
- 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 (CVP of Product for Developer Tools) and created the following thread in April 2019:

- Twitter thread: https://twitter.com/shahedC/status/1113177299652837376

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!
- Intellicode: 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 know 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 a recap of the launch 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=PLLPmfQOWse5Gr4pMcyFmwHW-hDUWmbT6B

So, what's new in the latest version of Visual Studio 2019 v16.6 Preview 2?

- **Version Control**: improved Git functionality, including quicker GitHub hosting with a single-click
- **Snapshot Debugging**: improved Snapshot Debugger experience, without requiring a restart when using Snapshot Debugger with Azure App Service
- **.NET Async:** new .NET Async tool (included with Performance Profiler) to help with better understanding and also optimizing your async/await code
- **JavaScript/TypeScript Debugging**: improved JS/TS debugger, with support for debugging service workers, web workers, iframes and in-page JS.
- **.NET Productivity**: multiple features for .NET developers including help for the following:
- add an explicit cast (when an implicit cast isn't possible),

- add a “file banner” across one or more code files,
- refactor/simply conditional expressions,
- convert regular string literals to verbatim strings

The list of features above is just a summary of the official announcement. For details on each of these features, check out the official announcement at:

- VS 2019 v16.6 Preview 2: https://devblogs.microsoft.com/visualstudio/visual-studio-2019-version-16-6-preview-2/

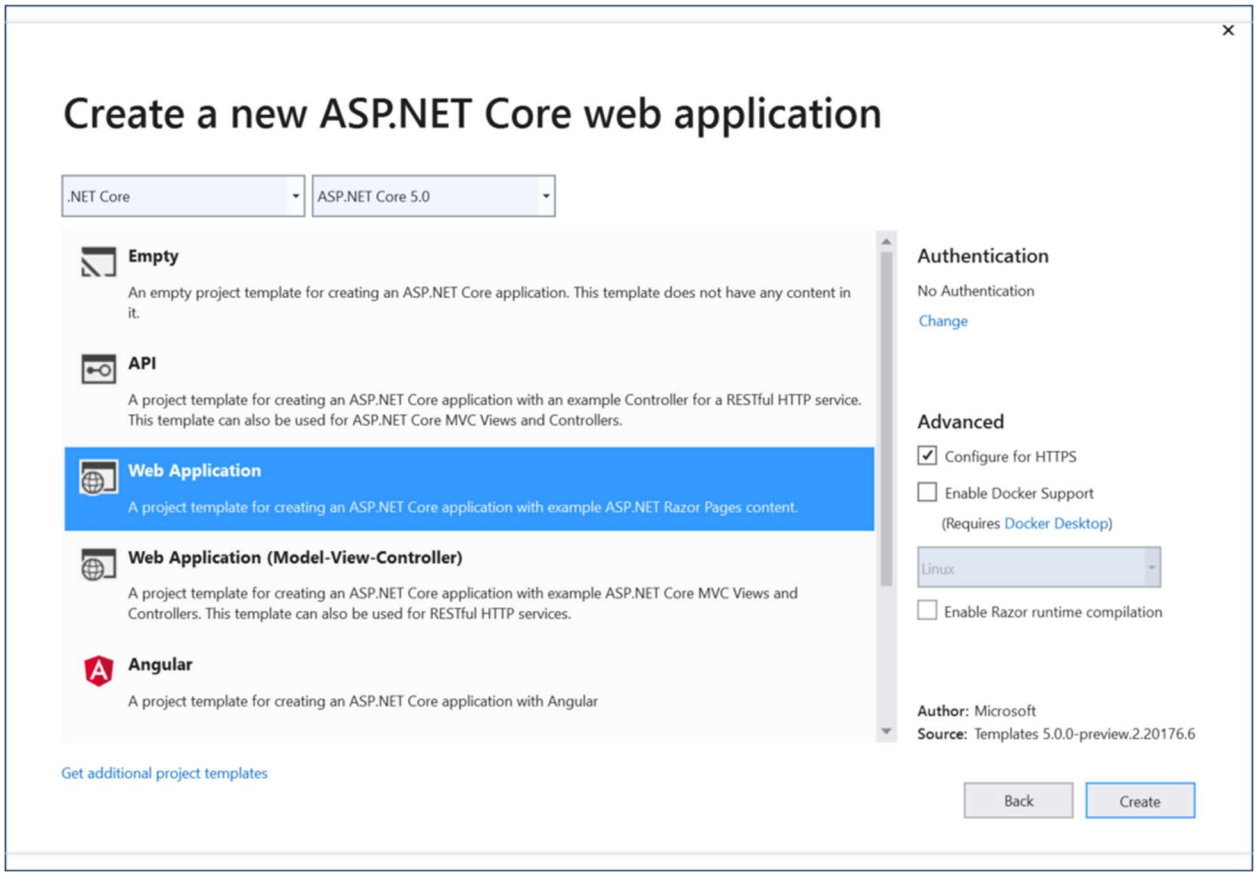
.NET 5.0 Preview 2

To get started, start with the latest version of .NET 5.0:

- Download .NET 5.0: https://dotnet.microsoft.com/download/dotnet-core/5.0

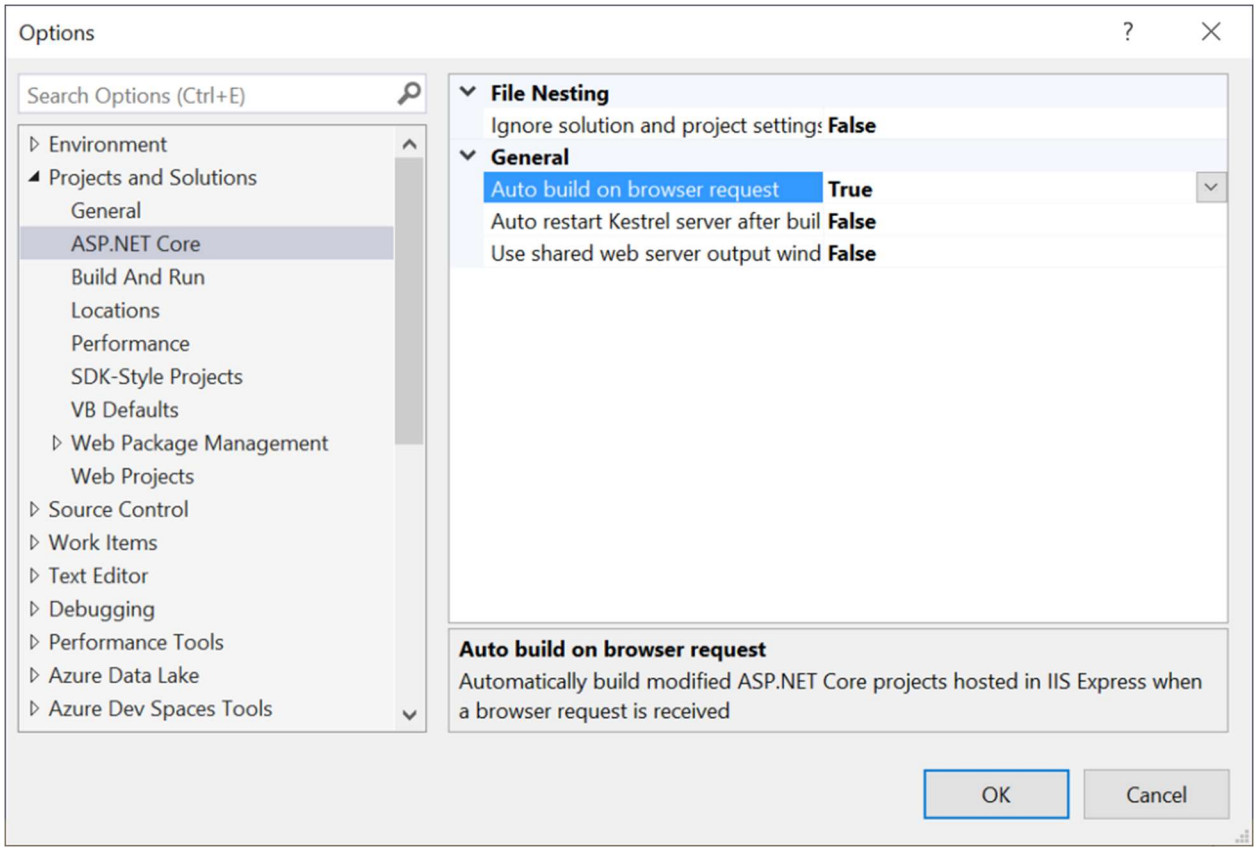
If you’ve already installed Visual Studio 2019, you may have followed the following steps:

1. Download Visual Studio 2019 Preview
2. Click **File** | **New** | **Project** (or create new from splash screen)
3. Create a new ASP .NET Core Web App
4. Select .NET Core and then ASP .NET Core 5.0 as the project type/platform



NOTE: For previous versions of .NET Core, you may not have seen an option for ASP .NET Core 3.0 in the list of project templates after installing VS2019. When ASP .NET Core 3.0 was still in preview as of April 2019 (even after the release of Visual Studio 2019), it was not automatically available for selection. In order to create ASP .NET Core 3.0 projects with VS2019, you would have had to install .NET Core 3.0 and enable previews of the .NET Core SDK in Visual Studio's Options panel.

In Visual Studio 2019 Preview, this additional step isn't necessary for ASP .NET Core 5.0. In fact, the Options panel includes settings for ASP .NET Core, without any mention of enabling preview versions.



VS2019 Options panel

C# 9.0 Features

To see what's to come in C# 9.0, check out the official list of milestones on GitHub, specifically Milestone 15 for C# 9.0:

- C# Language Milestones: https://github.com/dotnet/csharplang/milestones
- C# 9.0 Milestone 15: https://github.com/dotnet/csharplang/milestone/15

There are already dozens of feature candidates for the C# 9.0 release. The list includes Records and Pattern-Based "With" Expressions. From the official proposal, "*Records are a new, simplified declaration form for C# class and struct types that combine the benefits of a number of simpler features*".

This new feature allows you to omit an argument for its corresponding *optional* parameter, when invoking a function member. In this case, the value of the receiver's member is implicitly passed. This is accomplished in by the use of a "with-expression" and the use of this.*SomeIdentifier* to set a default argument.

Here is an example of what that could look like:

```
class TwitterUser
{
    public readonly int Followers;
    public readonly int Following;
    public TwitterUser With(
        int followers = this.Followers,
        int following = this.Following)
            => new TwitterUser(followers, following);

}
```

How would you use this so-called “caller-receiver default argument?

```
TwitterUser twitterUser1 = new TwitterUser(1000, 2000)
TwitterUser twitterUser2 = twitterUser1.With(followers: 0);
```

The above code creates a new TwitterUser, using the existing object to copy its values but changes the number of followers. For more information about the Records feature, check out the official proposal with detailed notes and code samples:

- C# Records Proposal: https://github.com/dotnet/csharplang/blob/master/proposals/records.md

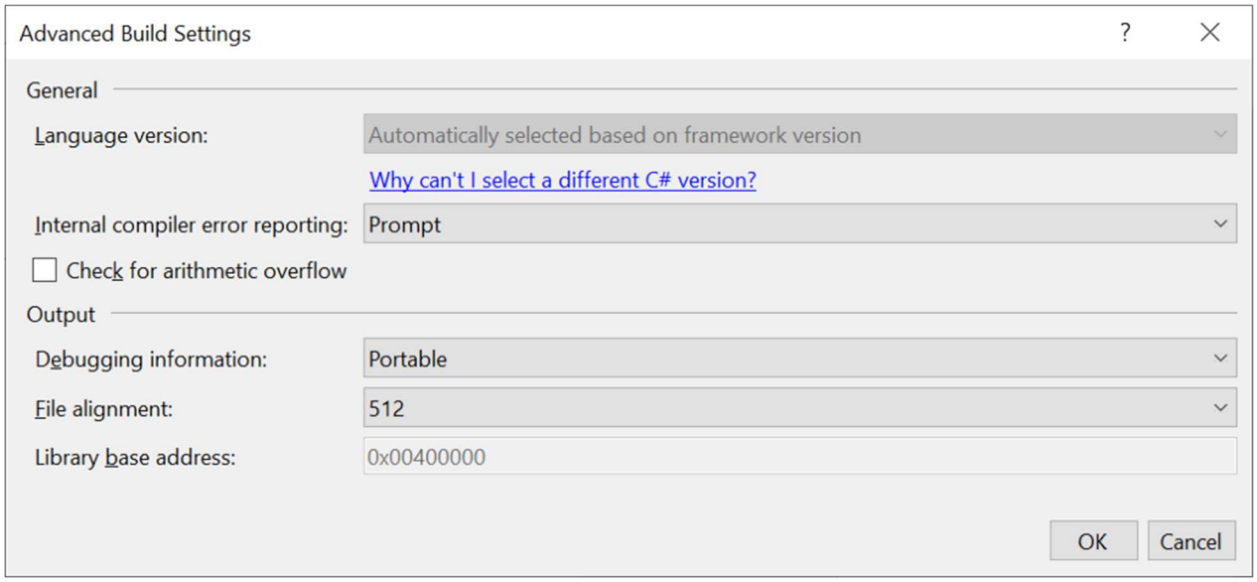
NOTE: To ensure that C# 9.0 preview features are available when a preview version becomes available, you can set the **LangVersion** property explicitly for your project. This setting is buried deep inside your Advanced settings within your project’s Build tab.

To update the language setting (after a future preview release):

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# 9.0 (beta)
6. ***Optional***: you may select “**unsupported preview**…” instead

If you try the above steps with VS2019 v16.6.0 Preview 2 (as of this writing), the language version selection is not available at this time. For more information on C# language versioning, check out the official documentation at:

- C# Language Versioning: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version



C# Language Version Selection

The above screenshot shows 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". This may be subject to change for C# 9 when it is released.

```
<Project Sdk="Microsoft.NET.Sdk">
   <PropertyGroup>
      <OutputType>Exe</OutputType>
      <TargetFramework>netcoreapp3.0</TargetFramework>
      <LangVersion>preview</LangVersion>
   </PropertyGroup>
</Project>
```

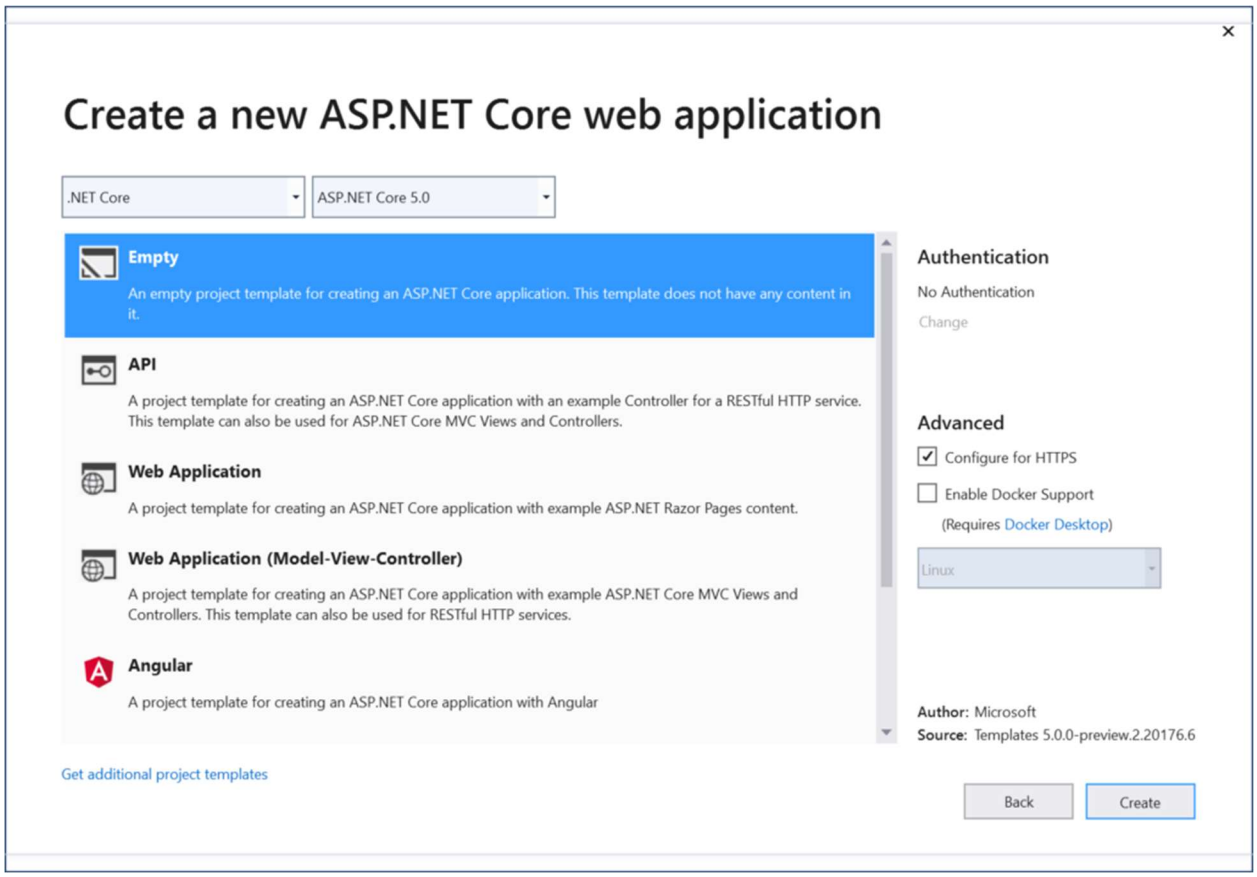
For a .NET Core 3.0 console app set to use C# 8.0 explicitly before its release, the value of <**LangVersion**> was set to the value "8.0". Once again, this may be subject to change for C# 9.

```
<Project Sdk="Microsoft.NET.Sdk">
   <PropertyGroup>
      <OutputType>Exe</OutputType>
```

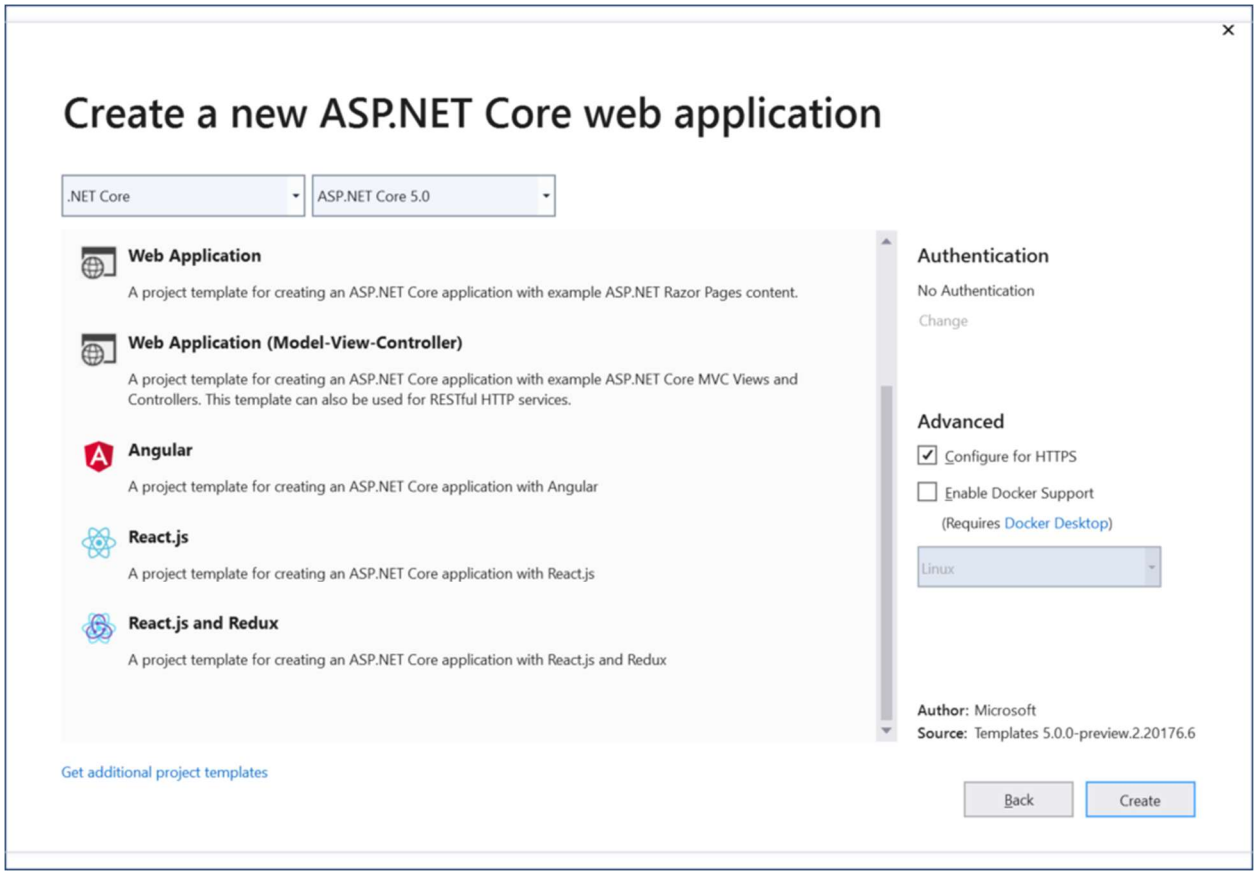
```
      <TargetFramework>netcoreapp3.0</TargetFramework>
      <LangVersion>8.0</LangVersion>
   </PropertyGroup>
</Project>
```

ASP .NET Core 5.0 Project Types

When you create a new .NET 5.0 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:



List of ASP .NET Core 5.0 projects



List of ASP .NET Core 5.0 projects (continued)

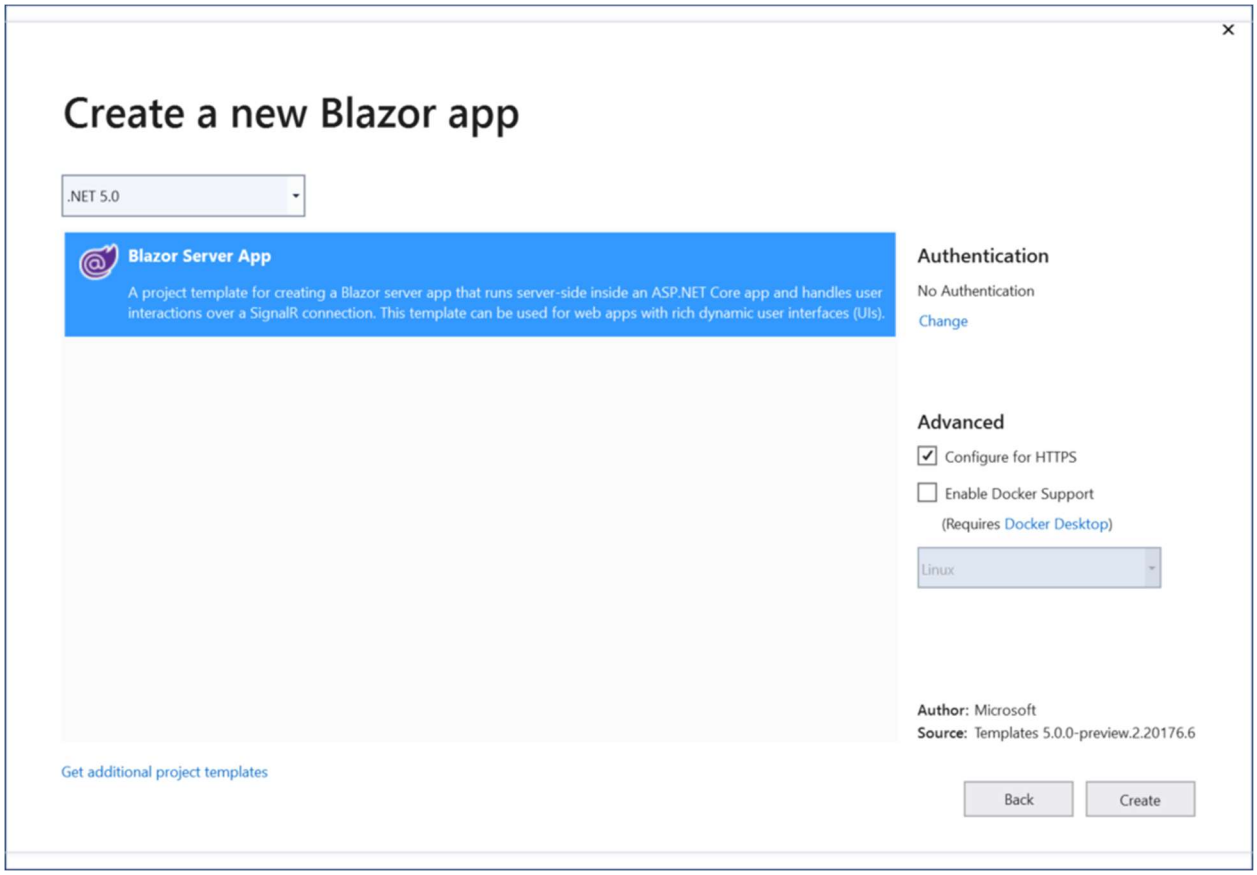
The above project types are described below:

1. **Empty**: familiar empty project that just writes out Hello World to the HTTP Response, *without* the use of MVC or Razor Pages
2. **API**: familiar project type for creating Web APIs and RESTful services. Can be mixed and matched with Razor Pages or MVC components.
3. **Web Application**: familiar project type for creating Web Apps with Razor Pages. Can be mixed and matched with Web API and/or MVC components.
4. **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.
5. **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.

FYI, the following project templates have moved out of their original location (inside ASP .NET Core web apps):

- **gRPC Service**: allows creation of a new gRPC service to make use of Google's high-performance Remote Procedure Call (RPC) framework
- **Worker Service**: allows creation of background processes, e.g. Windows services or Linux daemons.
- **Razor Class Library**: allows creation of reusable UI Class Libraries with Razor Pages. See previous post on Razor Class Libraries.

Blazor's server-side project template can also be found at the root-level of the template list, under Blazor App.



List of ASP .NET Core 5.0 projects (Blazor Server)

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 March 2020, client-side Blazor running in the browser with WebAssembly is still in preview (v3.2.0 Preview 3).

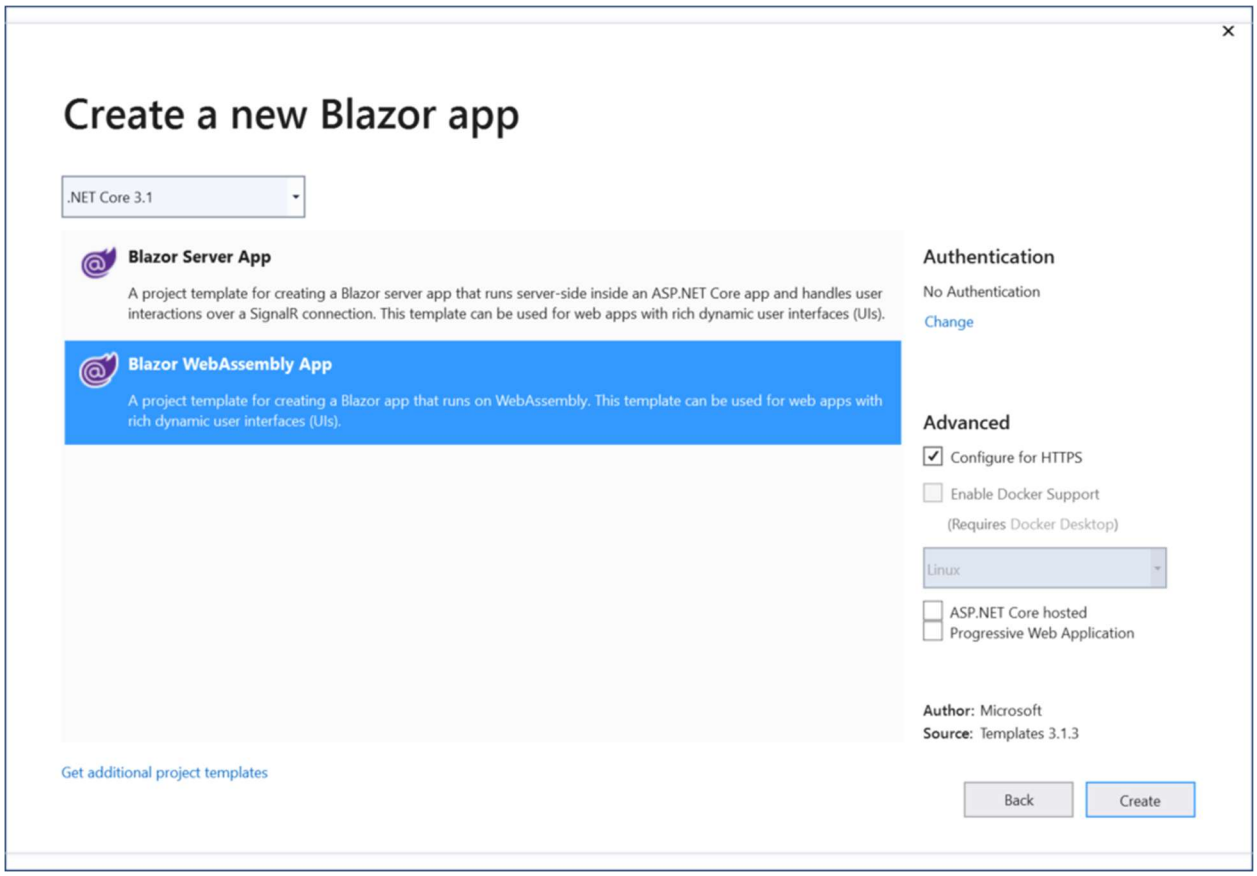
If you were to install the latest .NET Core 3.1 SDK, you would also have to download the latest Blazor WebAssembly separately.

- Install Blazor WebAssembly template: https://www.nuget.org/packages/Microsoft.AspNetCore.Components.WebAssembly.Templates/

In order to install the client-side template from a command line, run the following command:

dotnet new -i Microsoft.AspNetCore.Components.WebAssembly.Templates::3.2.0-preview2.20160.5

NOTE: Installing VS2019 v16.6 Preview 2 *automatically* installs an updated version of the .NET Core 3.1 SDK. This version includes the latest Blazor WebAssembly template. This means that a manual installation of client-side Blazor is not required when using this preview version of Visual Studio. However, you would have to select .NET Core 3.1 after selecting Blazor, if you wish to select the latest client-side Blazor WebAssembly template in VS2019 v16.6 Preview 2.



Blazor templates in VS2019 Preview

On a related note, check out my previous post on *server-side* Blazor:

- Blazor Full-Stack Web Dev in ASP .NET Core 3.1: https://wakeupandcode.com/blazor-full-stack-web-dev-in-asp-net-core-3-1/

Migrating from 3.1 to 5.0

With all this goodness on the roadmap, you may be wondering how you can start migrating your ASP .NET Core 3.1 projects to .NET 5.0. Fear not, the official guide (a work in progress) is now available:

- Migrate from ASP .NET Core 3.1 to 5.0: https://docs.microsoft.com/en-us/aspnet/core/migration/31-to-50

This documented migration process involves the following steps:

1. Update .NET Core SDK version in global.json (from “3.1.200” to “5.0.100-preview.2.20176.6”)
2. Update the target framework (from netcoreapp3.1 to netcoreapp5.0)
3. Update package references (e.g. from “3.1.2” to “5.0.0-preview.2.20167.3”)
4. Update Docker images to include a base image that includes ASP .NET Core 5.0 (e.g. by using the docker pull command followed by mcr.microsoft.com/dotnet/core/aspnet:5.0)
5. Review breaking changes: refer to https://docs.microsoft.com/en-us/dotnet/core/compatibility/3.1-5.0

To deploy a .NET 5 project via Azure DevOps, try the following YAML

```
steps:
- task: UseDotNet@2
  inputs:
    packageType: sdk
    version: 5.0.100-preview.2.20176.6
    installationPath: $(Agent.ToolsDirectory)/dotnet
```

The above YAML snippet is a suggestion from .NET developer Neville Nazerane via the unofficial ASP .NET Core Facebook group:

- YAML snippet source: https://www.facebook.com/groups/about.asp.net.core/permalink/1538257476352234/?comment_id=1539064492938199

NOTE: ASP .NET Core in .NET 5.0 Preview 2 only includes minor bug fixes (but not any major new features).

References

- [May 6, 2019] Introducing .NET 5: https://devblogs.microsoft.com/dotnet/introducing-net-5/
- [March 16, 2020] Announcing .NET 5.0 Preview 1: https://devblogs.microsoft.com/dotnet/announcing-net-5-0-preview-1/
- [April 2, 2020] Announcing .NET 5.0 Preview 2: https://devblogs.microsoft.com/dotnet/announcing-net-5-0-preview-2/
- ASP .NET Core Updates in .NET 5 Preview 2: https://devblogs.microsoft.com/aspnet/asp-net-core-updates-in-net-5-preview-2/
- VS 2019 v16.6 Preview 2: https://devblogs.microsoft.com/visualstudio/visual-studio-2019-version-16-6-preview-2/
- EF Core 5.0 Preview 2: https://devblogs.microsoft.com/dotnet/announcing-entity-framework-core-5-0-preview-2/
- Migrate from ASP .NET Core 3.1 to 5.0: https://docs.microsoft.com/en-us/aspnet/core/migration/31-to-50
- C# 9 Language Features: https://github.com/dotnet/csharplang/milestone/15
- Blazor WebAssembly 3.2.0 Preview 3: https://devblogs.microsoft.com/aspnet/blazor-webassembly-3-2-0-preview-3-release-now-available/
- Blazor WebAssembly Template on NuGet: https://www.nuget.org/packages/Microsoft.AspNetCore.Components.WebAssembly.Templates/

Organizational Accounts for ASP .NET Core 3.1

By Shahed C on April 13, 2020

5 Replies

This is the fifteenth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- .NET 5.0, VS2019 Preview and C# 9.0 for ASP .NET Core developers

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.15-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.15-alpha

NOTE: The NetLearner suite of apps won't be updated to use organizational authentication in the main branch, so you can check out the new sample code in the experimental subfolder, merged from a branch:

- Org Auth: https://github.com/shahedc/NetLearnerApp/tree/main/experimental/NetLearner.OrgAuth

In this Article:

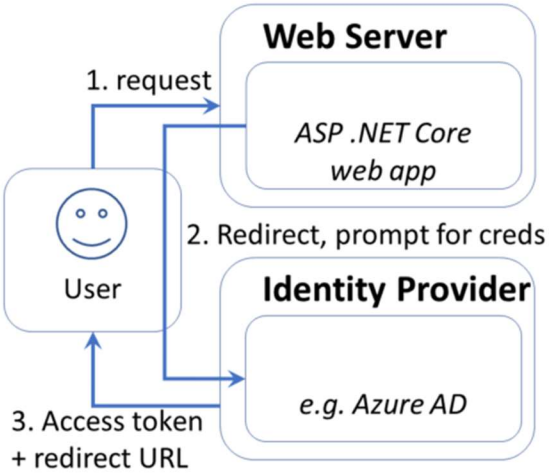
O is for Organizational Accounts

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 (accounts stored in-app).
- **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. This authentication process is illustrated in the diagram shown below:



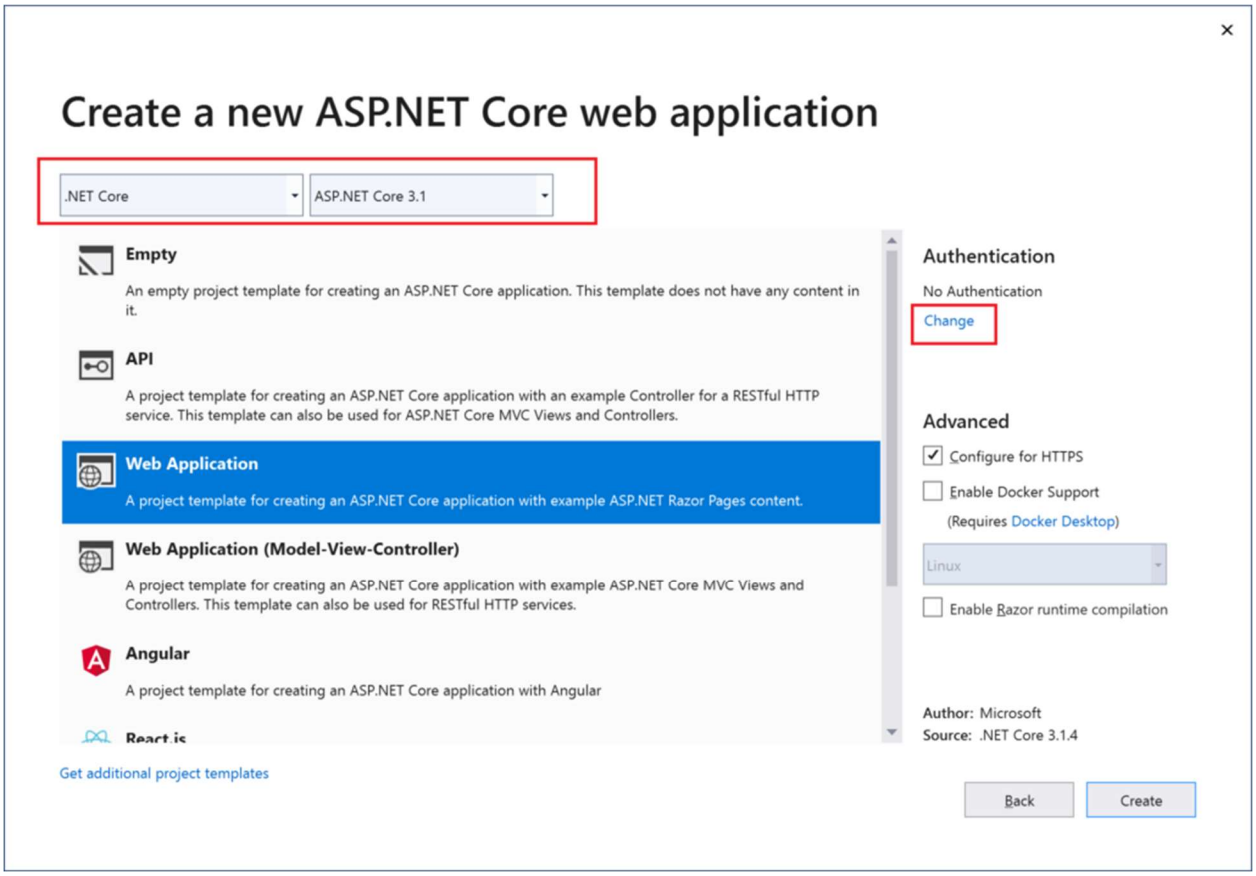
ASP .NET Core web authentication via AzureAD

Adding Authentication

To add authentication to a new project quickly, here are the instructions for Visual Studio 2019.

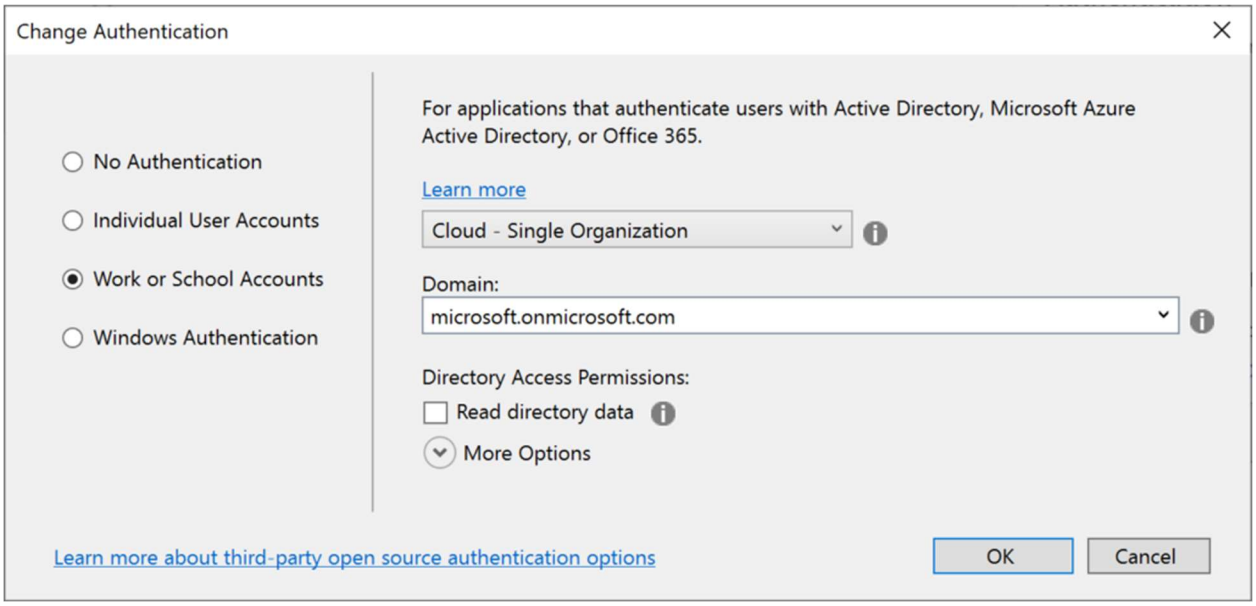
If you choose to use the new splash screen:

1. Click “Create a new project”
2. Select “ASP .NET Core Web Application” or “Blazor” (server-side)
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 and then ASP .NET Core 3.1 from dropdowns
8. Select a project type, e.g. Empty, Web Application (Razor Pages or MVC), Blazor App.
9. Click the “Change” action link in the Authentication section



Creating a new web app project with authentication

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.



Work/School account selected for authentication method

NOTE: If you need help creating new .NET Core 3.1 project types in Visual Studio, take a look at this previous blog post in this series on .NET Core 3.0 to get some help on how to enable it.

- Hello ASP .NET Core 3.1: https://wakeupandcode.com/hello-asp-net-core-v3-1/

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

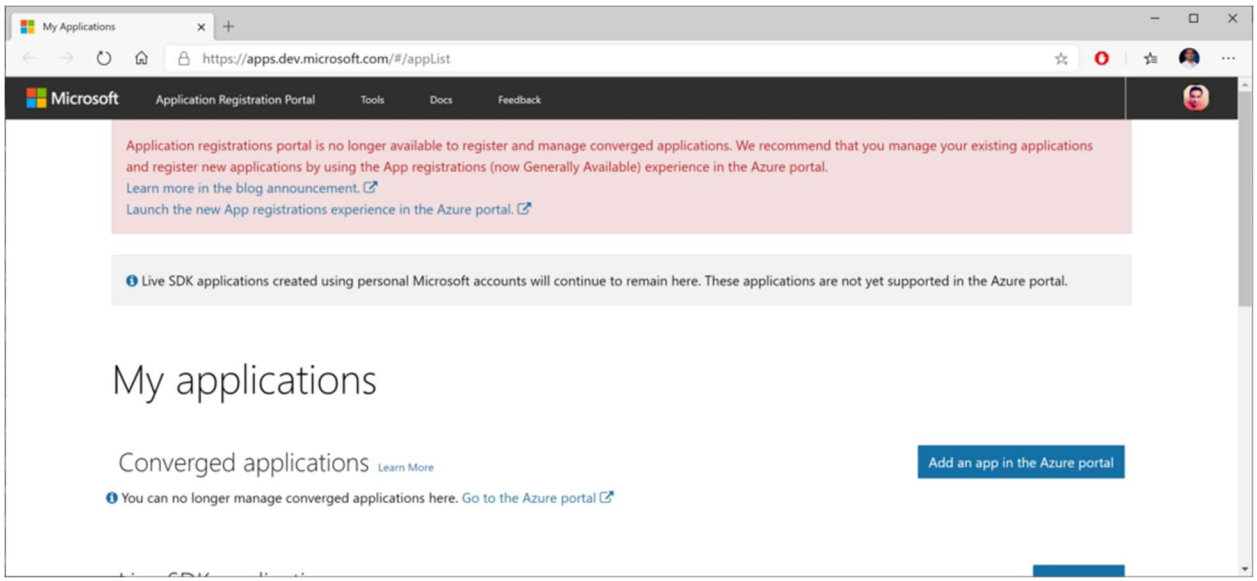
To use CLI Commands in a Terminal window, use the **dotnet new** command followed by the --auth flag. The authentication type can be any of the aforementioned authentication types, e.g. Individual.

```
dotnet new mvc --auth Individual -o myproj
```

Configuring App Registration

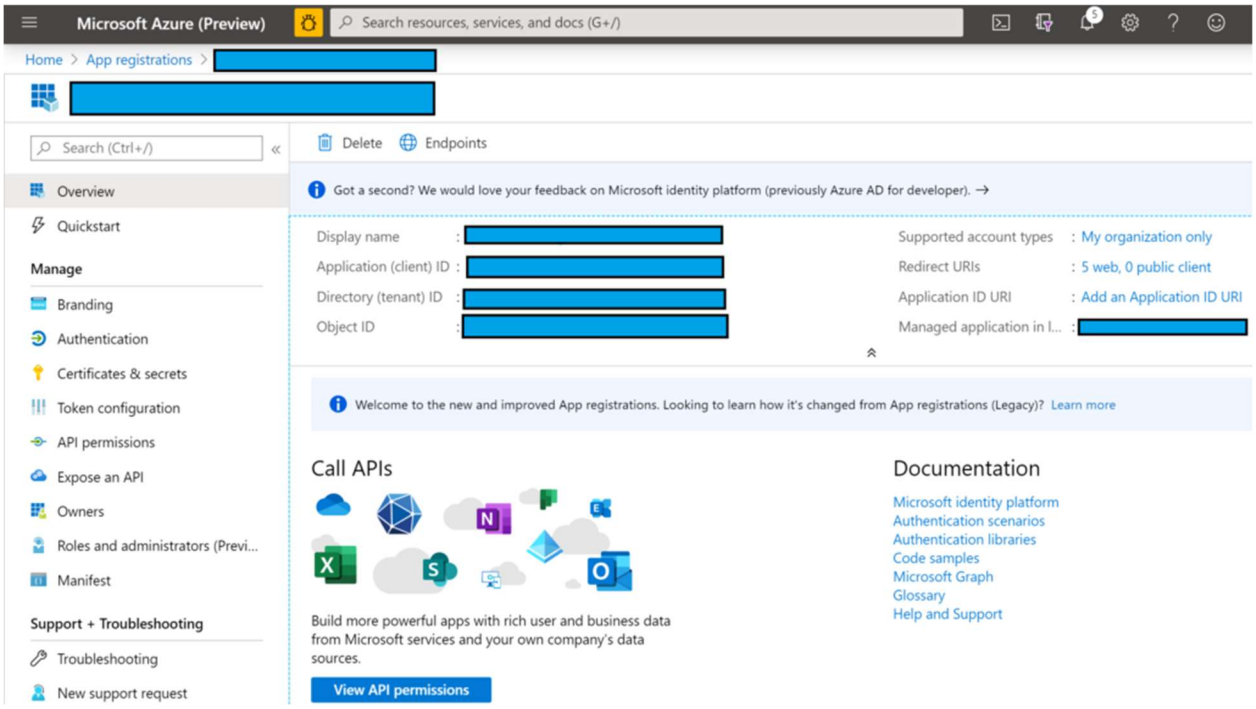
If you've used "Individual User Accounts" before, you've probably used a database to store user data, either on-premises or in the cloud. If you've used "Work or School Accounts" (i.e. organizational accounts), you may have the old App portal at the following URL:

- (OLD) Application Registration Portal: https://apps.dev.microsoft.com



Old App Registration Portal

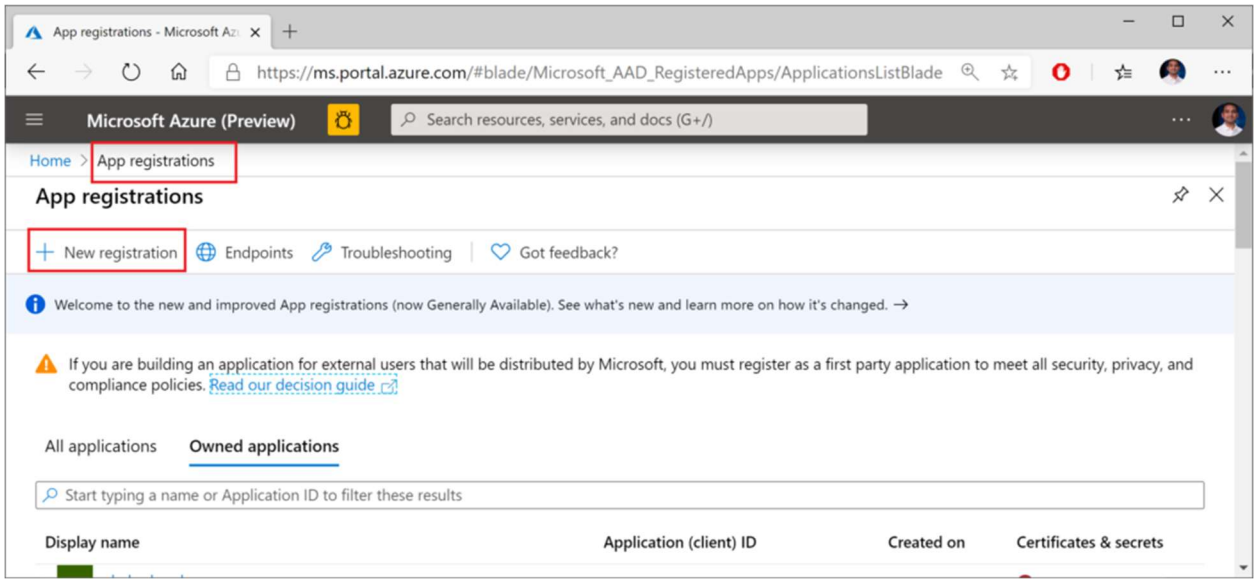
On the old site, you may see a message suggesting that you should go to the Azure Portal to use the newer (current) App Registrations feature. This feature was previously in preview, and you can currently use it directly in the Azure Portal. If you click the link from the old site, it should take you directly to the App Registrations page, and may prompt you to log in first.



App Registration screen showing overview

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.

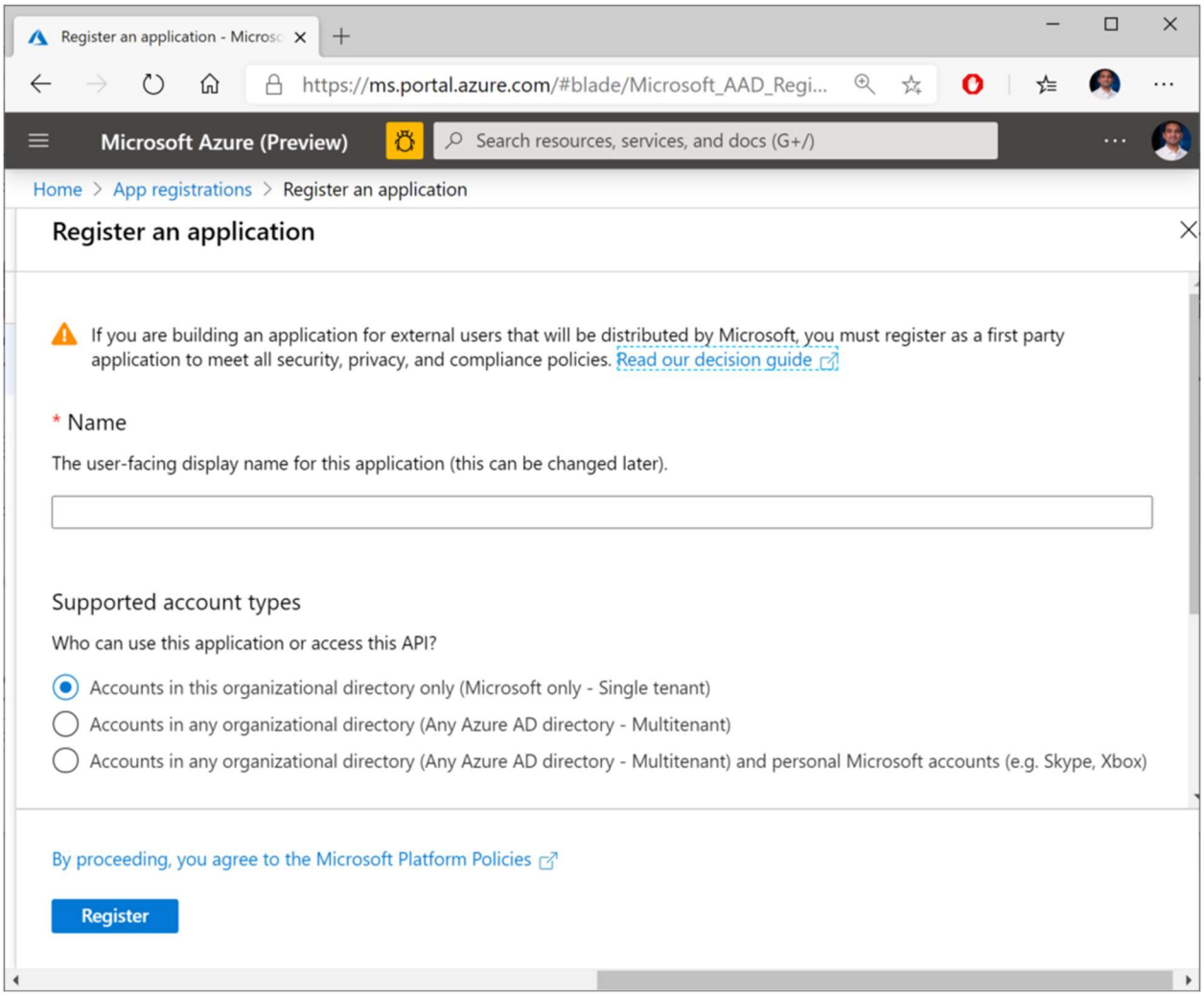
1. Log in to the Azure Portal
2. Search for App Registrations
3. Arrive at the App Registrations page
4. When a newer preview version becomes available, you may have to click the banner that takes you to a preview experience.



New App Registration in Azure Portal

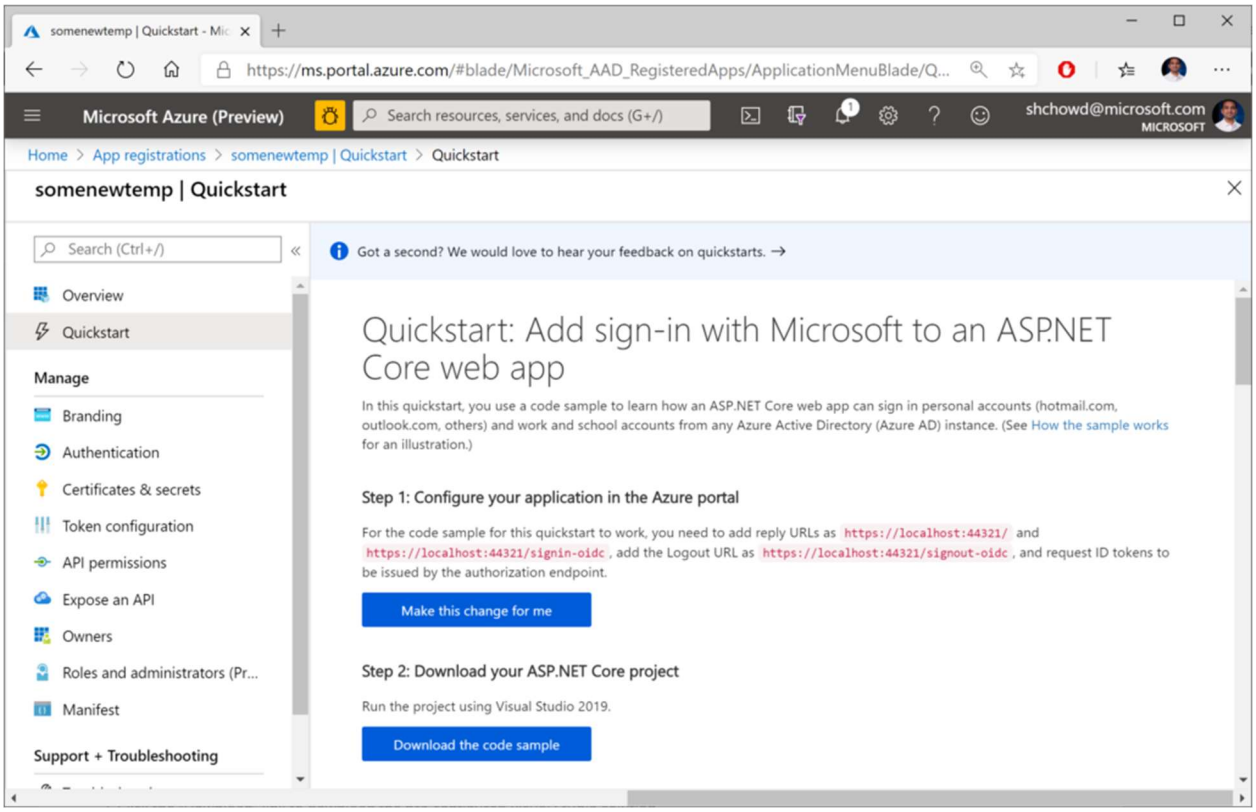
In the form that follows, fill out the values for:

- **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 for authentication

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.



Quickstart screen for app registration

In the steps that follow:

1. Click the “Make the changes for me” button to make the necessary configuration changes.
2. Optional: Click the “Download” link to download the pre-configured Visual Studio solution, if you don’t already have a project.

At the time of this writing, the downloadable sample project is a VS2019 application for ASP .NET Core 2.2. You can download it to inspect it, but I would recommend creating a new project manually in VS2019. There may be 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**(). For different project types (MVC, Razor Pages, Blazor), you should also see some additional code setting up the authorization policy.

```
// contents of ConfigureServices() when created in VS2019

services.AddAuthentication(AzureADDefaults.AuthenticationScheme)
.AddAzureAD(options => Configuration.Bind("AzureAd", options));
```

The string value “AzureAd” is referenced in your appsettings file, e.g. appsettings.json in MVC, Razor Pages, or Blazor projects. The properties for this entry include the following:

- **Instance**: URL for login page, e.g. https://login.microsoftonline.com
- **Domain**: your AD tenant’s domain, e.g. microsoft.onmicrosoft.com
- **TenantId**: your Tenant Id value, usually a GUID
- **ClientId**: Your app’s Client Id, obtained from app registration
- **Callback Path**: partial path for sign-in, e.g. /signin-oidc

```
{
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "Domain": "<REPLACE_DOMAIN_NAME>",
    "TenantId": "<REPLACE_TENANT_ID>",
    "ClientId": "<REPLACE_CLIENT_ID>",
    "CallbackPath": "/signin-oidc"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

You can see more information on appsettings.json values in the official docs. You can also open the apssettings.json from the portal-downloaded project to get your own app's Id values. The following documentation is specifically written for v2, but offers explains 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 get a refresher on Authentication and Authorization in ASP .NET Core 3.1, check out the following post on from earlier in this blog series.

- Authentication and Authorization in ASP .NET Core 3.1: https://wakeupandcode.com/authentication-authorization-in-asp-net-core-3-1/

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 in an MVC or Razor Pages web app. Here is a snippet from the samples:

```
@if (User.Identity.IsAuthenticated)
{
 <li class="nav-item">
 <span class="nav-text text-dark">Hello @User.Identity.Name!</span>
 </li>
...
}
```

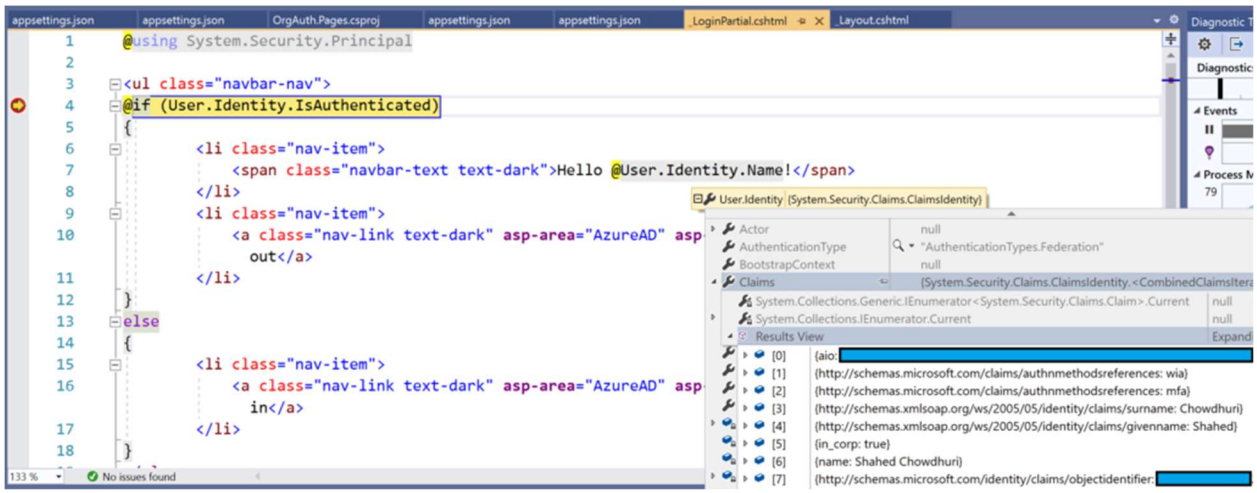
NOTE: The Blazor web app wraps the authorized content in an <**Authorized**> tag within an <**AuthorizedView**. For more on the Blazor project, take a look at the Blazor post:

- Blazor full-stack web dev: https://wakeupandcode.com/blazor-full-stack-web-dev-in-asp-net-core-3-1/

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

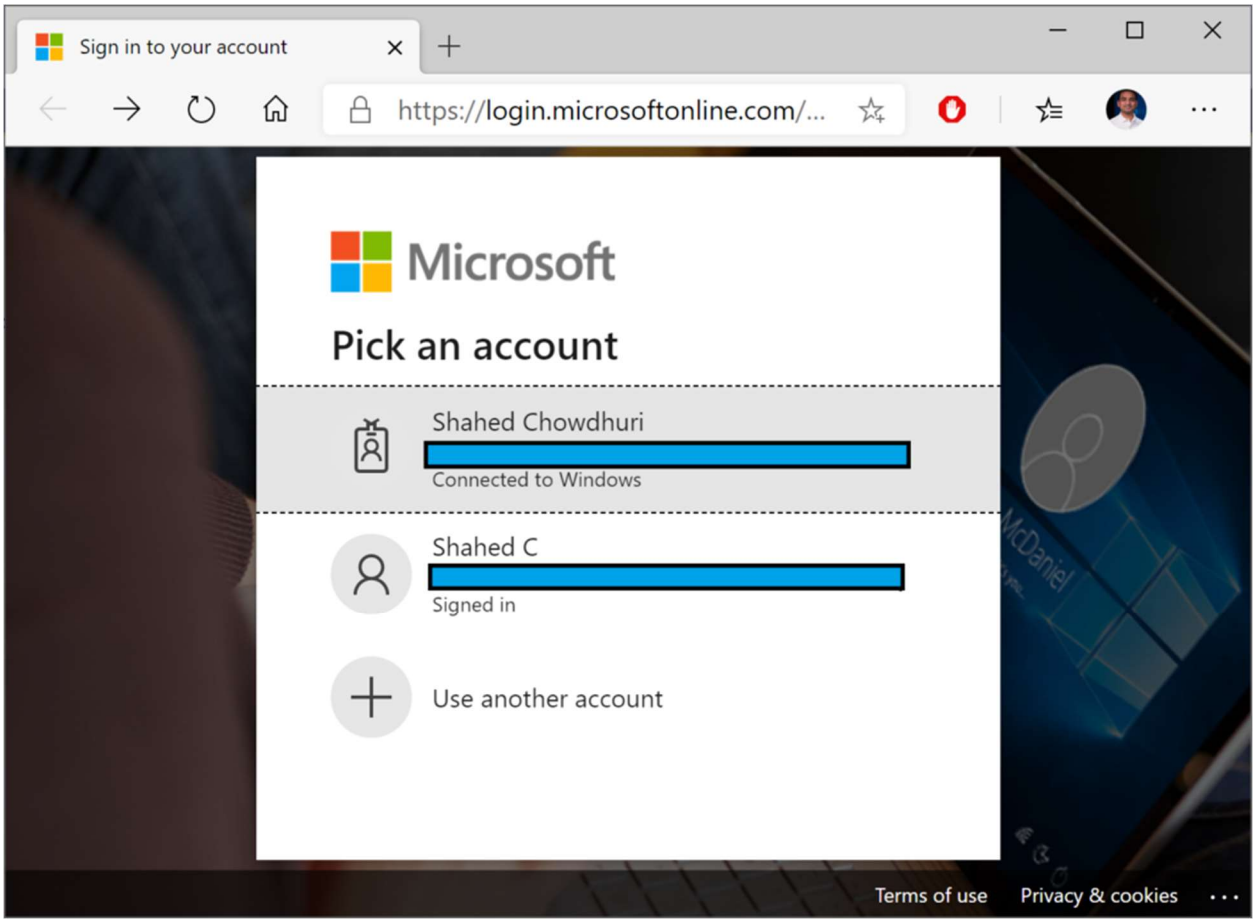
The following screenshot shows an example of the user information in my debugging environment when logged in:



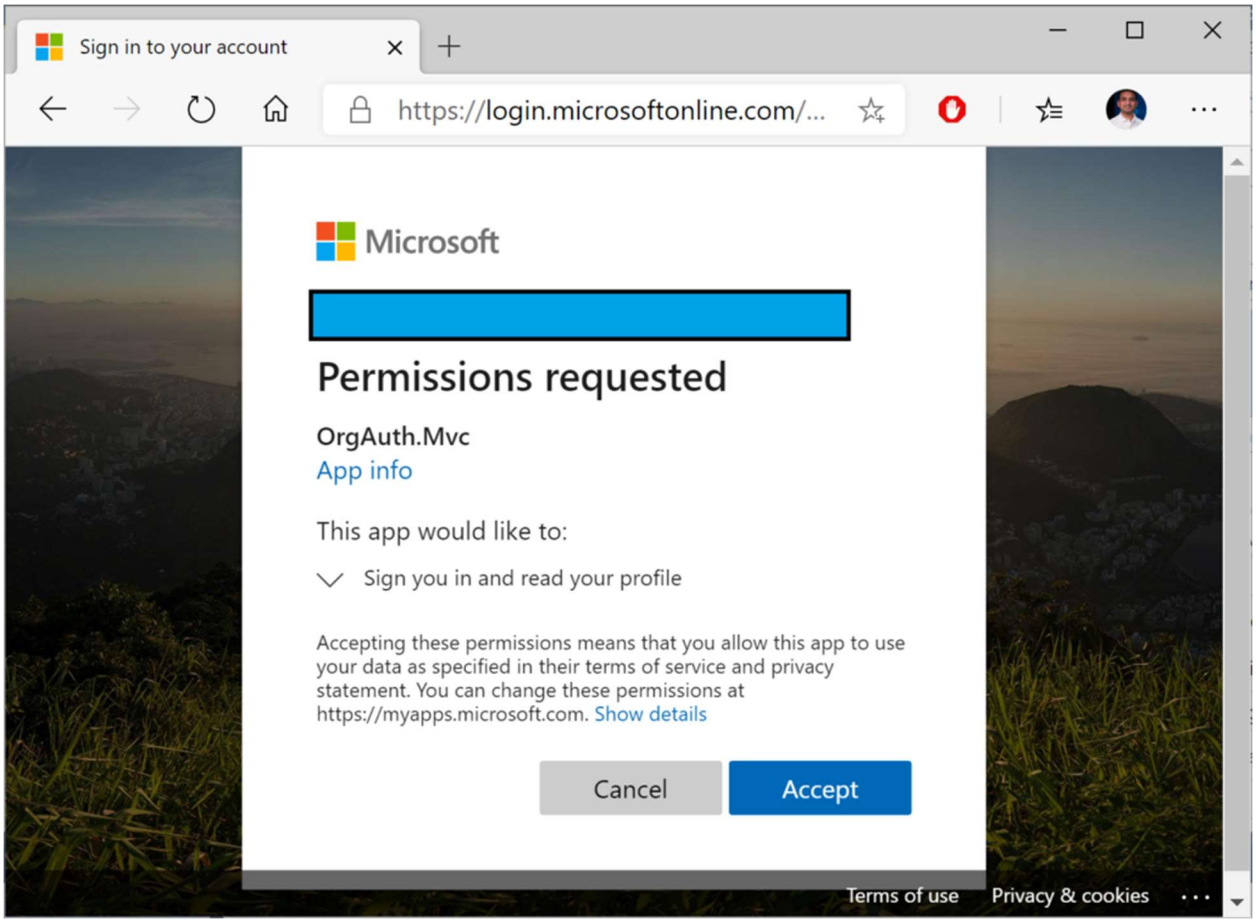
User.Identity.Name with Claims in debugger

Running the Samples

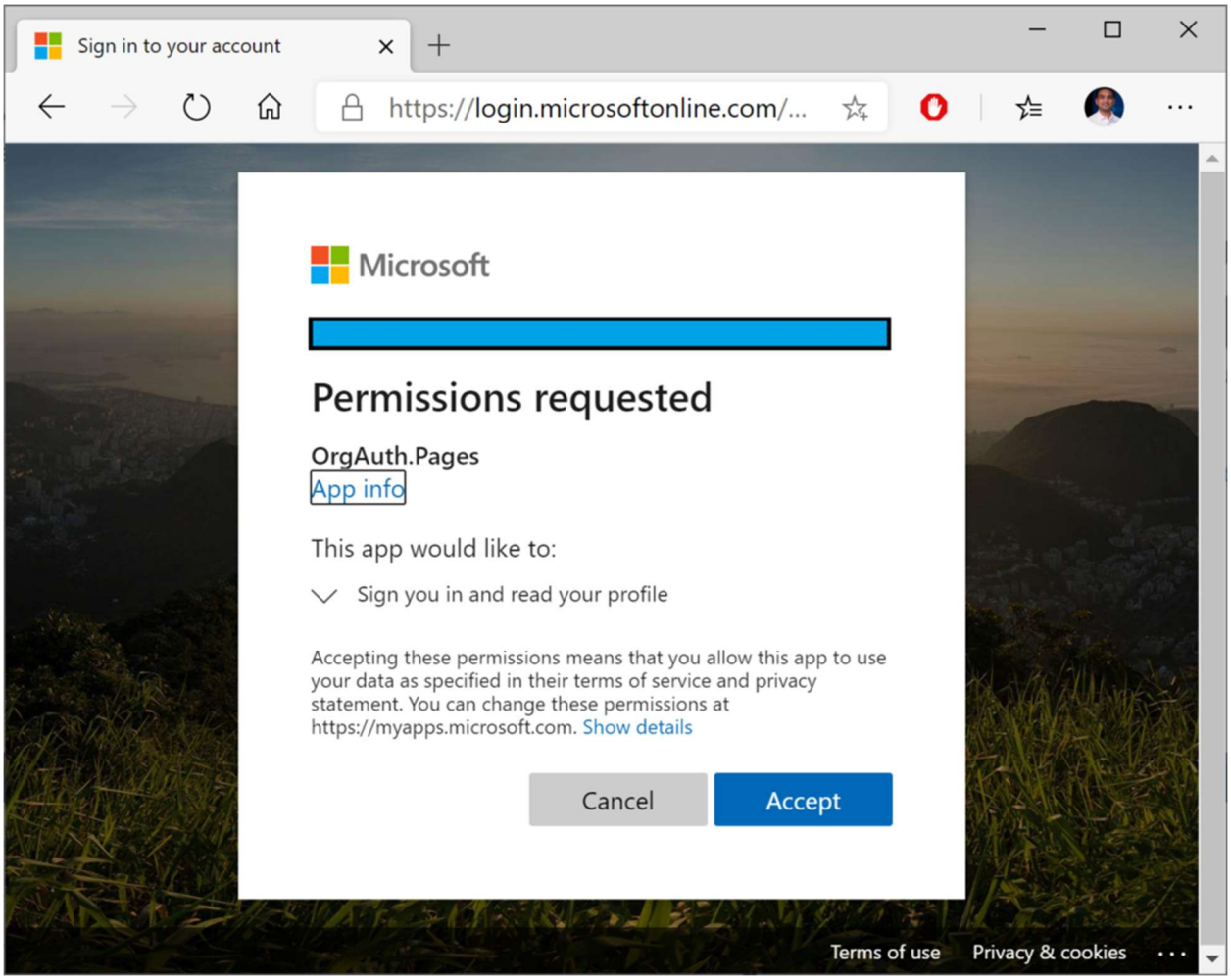
Running the sample web projects should redirect you to the Azure AD login page for your tenant. The login page should look identical for all 3 project types: MVC, Razor Pages or Blazor. If you've already logged in to one or more accounts in your browser, you should have one or more identities to select from.



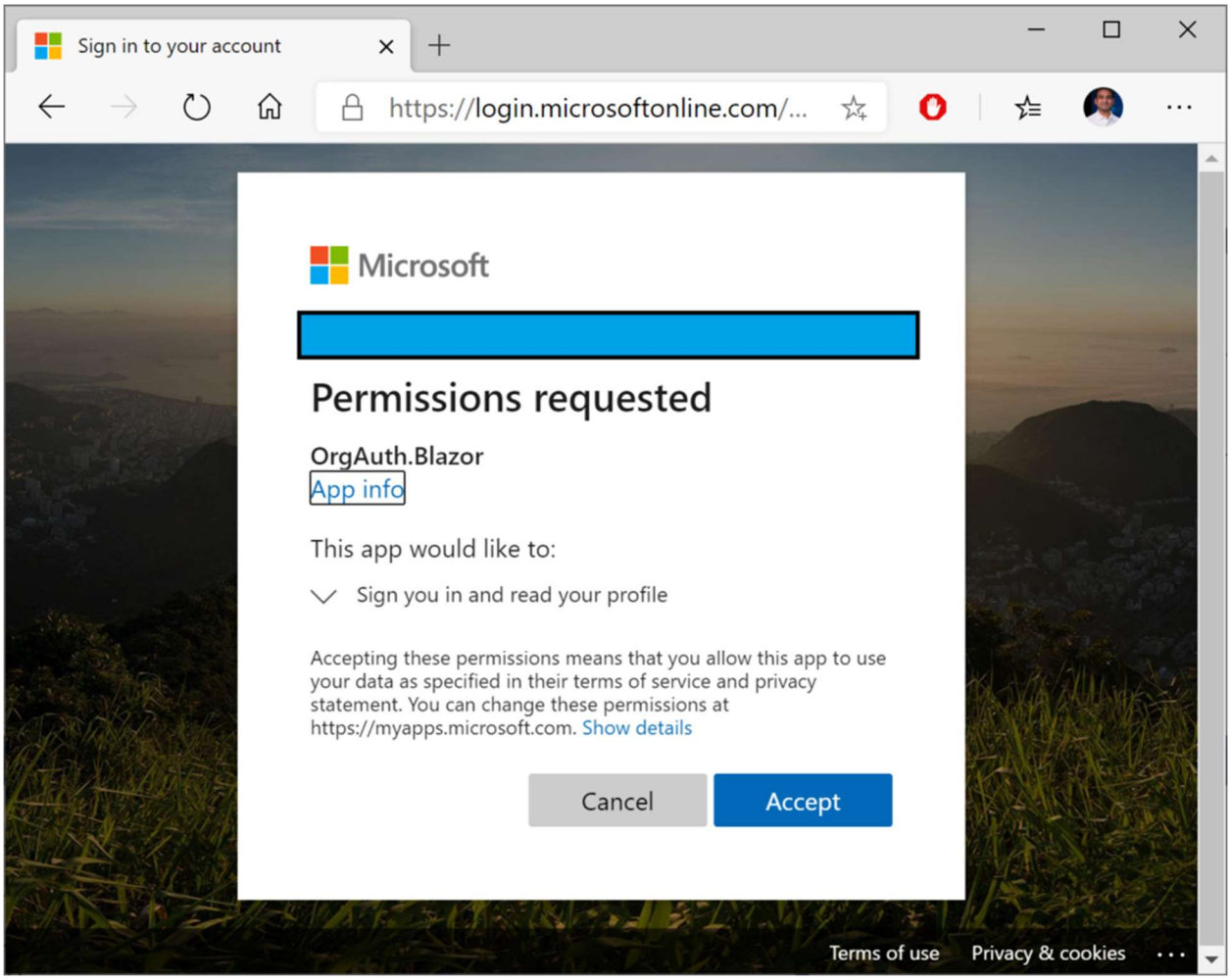
Once you've selected an account, you should see a confirmation screen (at least the first time) to Accept the authentication and login. This screen will look similar for all your projects, and the project name should be displayed in the middle.



MVC app authentication confirmation

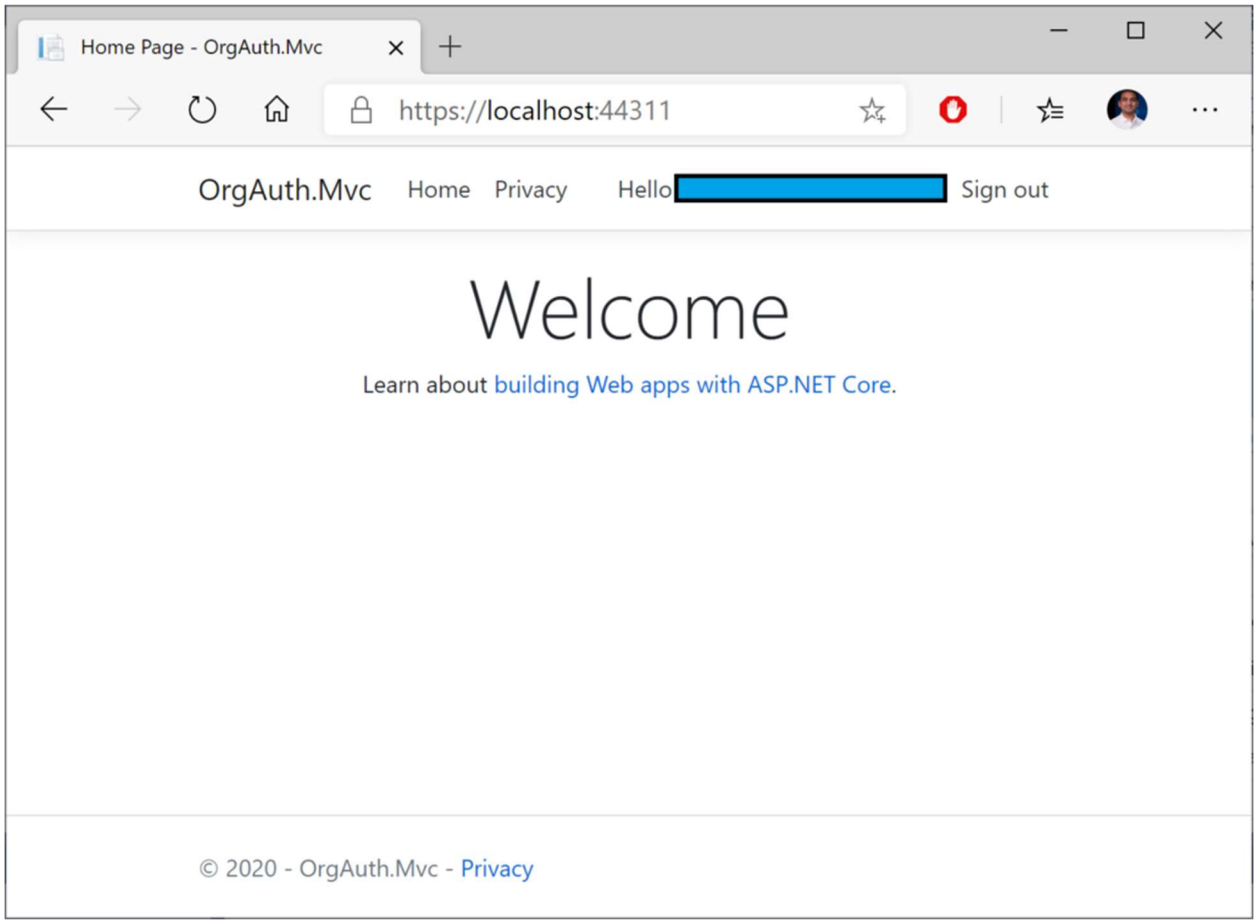


Razor Pages app authentication confirmation

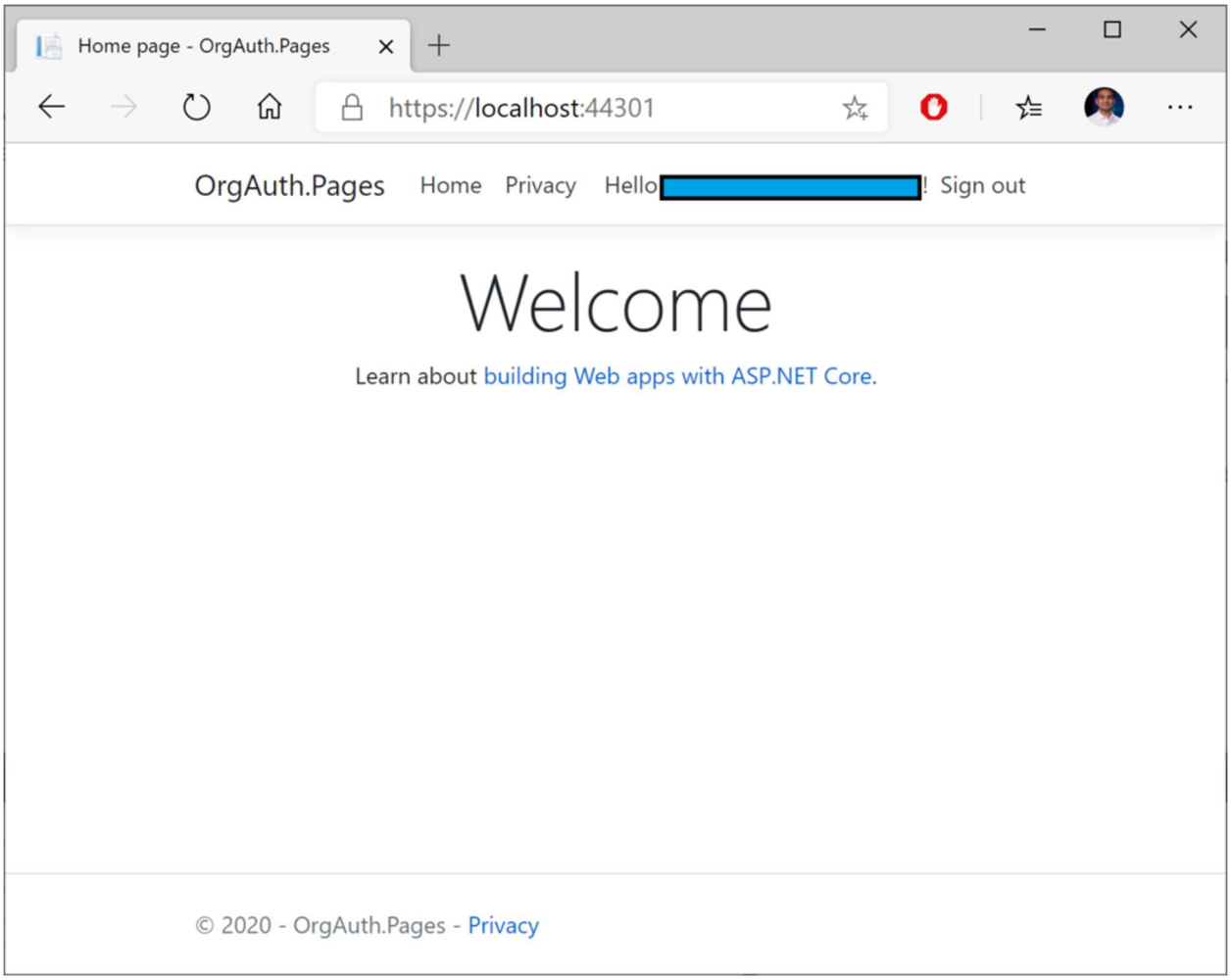


Blazor app authentication confirmation

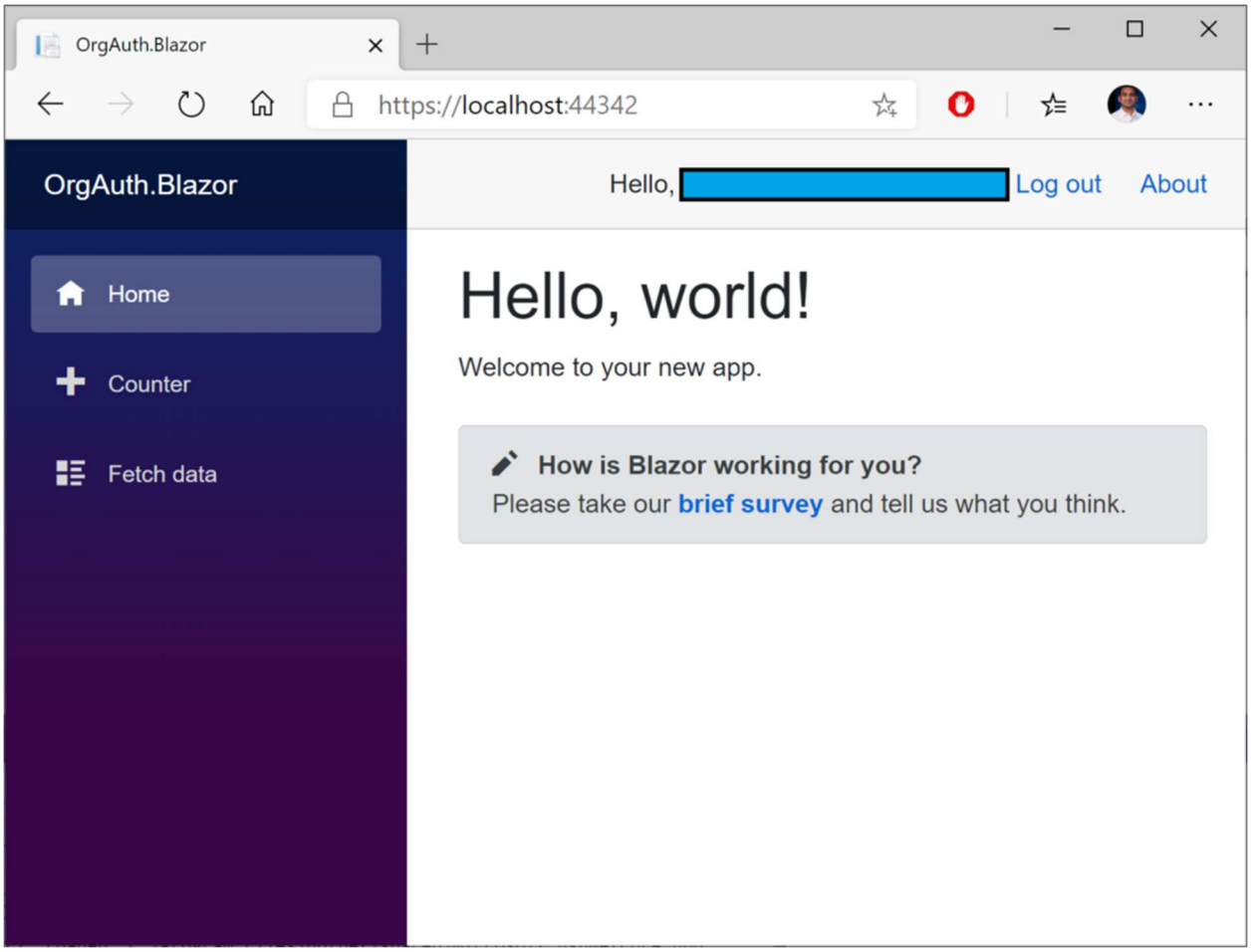
Finally, you should see each web application with some indication that you are currently logged in. In the template-generated sample apps, this is demonstrated by the “Hello” message in the header area.



MVC web app, logged in



Razor Pages web app, logged in



Blazor web app, logged in

References

- 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 3.1 Web Apps

By Shahed C on April 20, 2020

5 Replies

This is the sixteenth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Organizational Accounts for ASP .NET Core 3.1

NetLearner on GitHub:

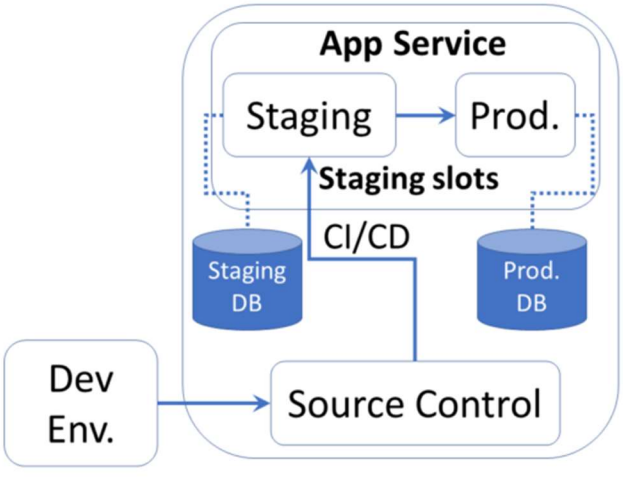
- Repository: https://github.com/shahedc/NetLearnerApp
- v0.16-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.16-alpha

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 productions 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.



Server Environments from Dev to Prod via Staging

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 publish a web app directly to production or worrying about downtime, you can publish to a Staging Slot and the perform a “swap” operation to essentially promote your Staging environment into Production.

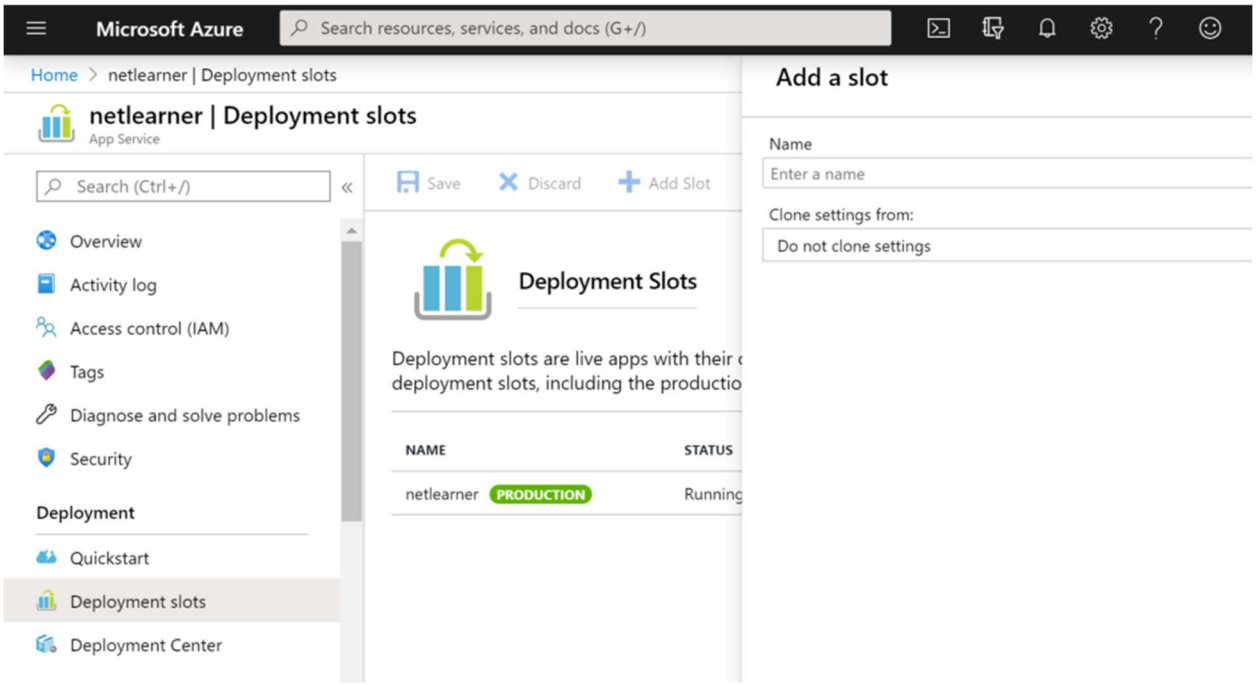
NOTE: To enable multiple deployment slots in Azure, you must be using an App Service in a Standard, Premium, or Isolated tier.

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 3.1 to Azure App Service: https://wakeupandcode.com/deploying-asp-net-core-3-1-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, chose a source to clone settings (or not)
7. Click the Add button to create a new slot



Deployment Slots in Azure App Service

Once you set up multiple slots for staging and production, you may use the Swap feature to swap your deployed application 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.

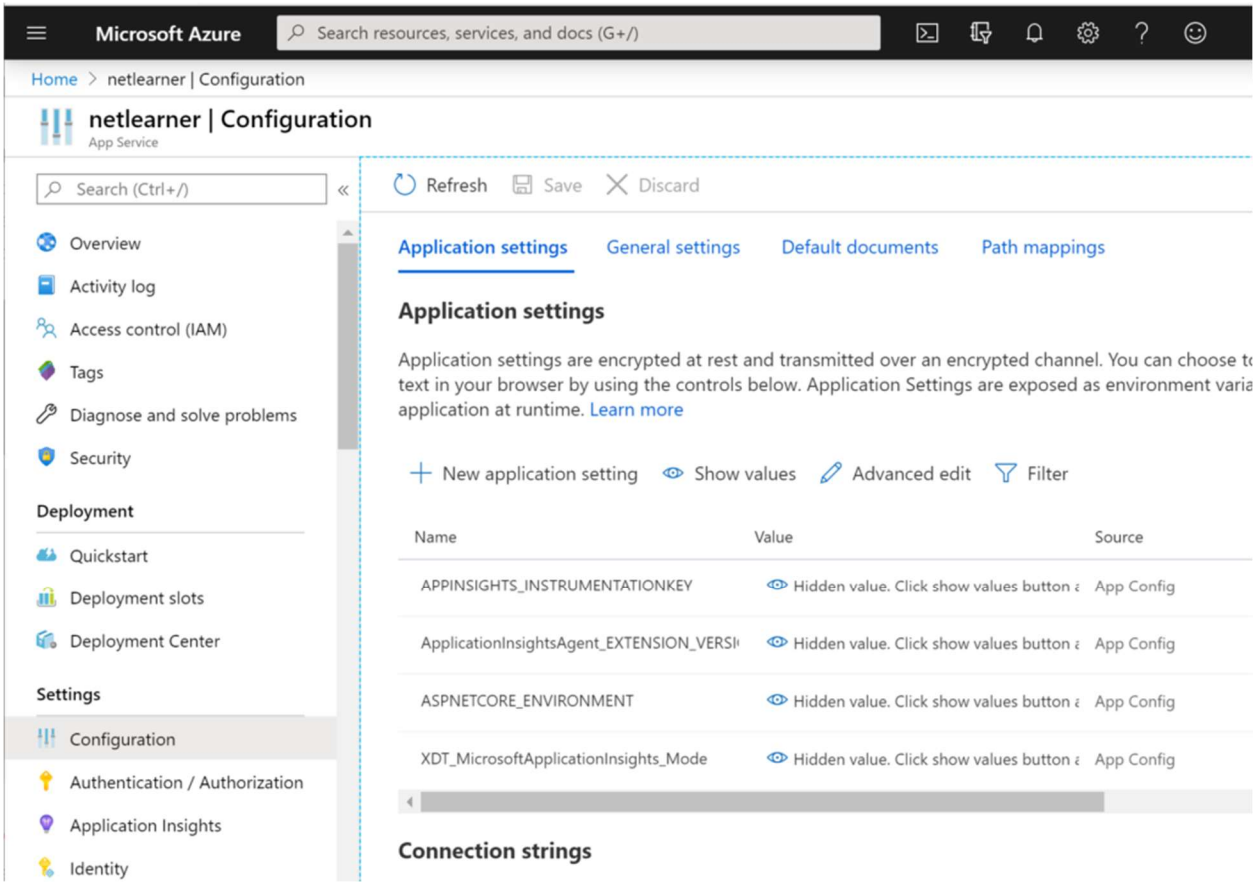
When deploying your application through various means (Visual Studio Publish, Azure CLI, CI/CD from your Source Control System, etc), you may choose the exact slot when there is more than one. You may also set up “Auto-Swap” to swap a slot (e.g. staging) automatically into production, upon pushing code 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.



Configuration screen in Azure App Service

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 2020 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-3-1-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-3-1/

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:

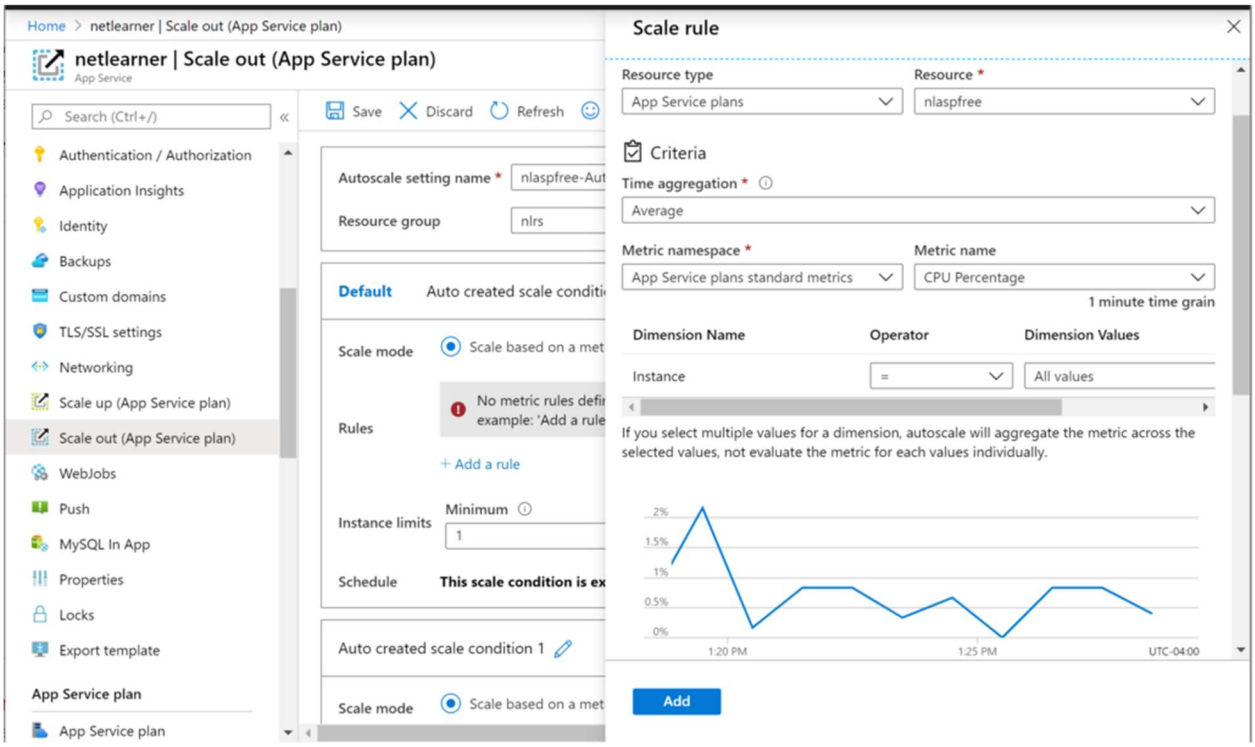
```
Script-Migration -Output "myMigrations.sql"
```

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 minmium 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.



Adding a Scale Rule to Scale Out in Azure App Service

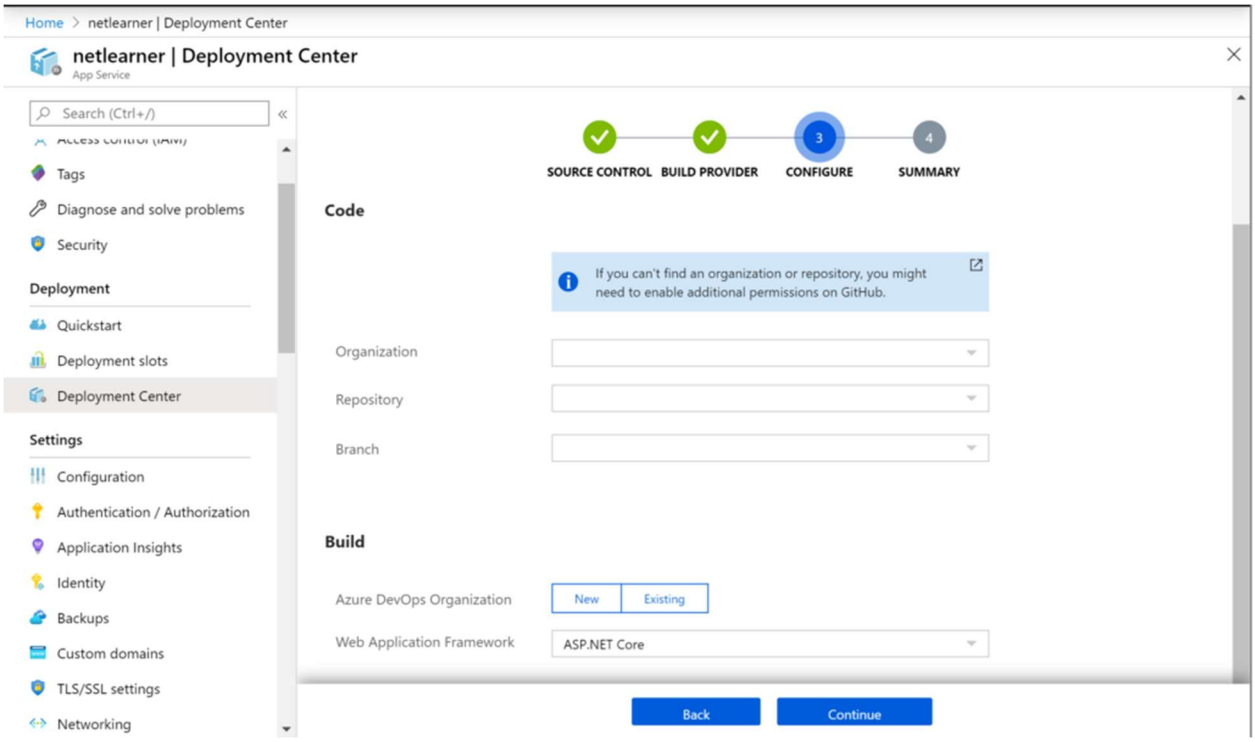
NOTE: In order to make use of Auto-Scaling, you'll have to upgrade to the appropriate tier to do so. You can still use Manual Scaling at a lower tier. Scalability features are not available on the F1 Free Tier.

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 relatively-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)



Deployment Center in Azure App Service

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-3-1/
- Handling Errors in ASP .NET Core: https://wakeupandcode.com/handling-errors-in-asp-net-core-3-1/

For more information on *Application Insights*, check out the official documentation at:

- App Insights Overview: https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview

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 3.1 Web Apps

By Shahed C on April 27, 2020

3 Replies

This is the seventeenth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Production Tips for ASP .NET Core 3.1 Web Apps

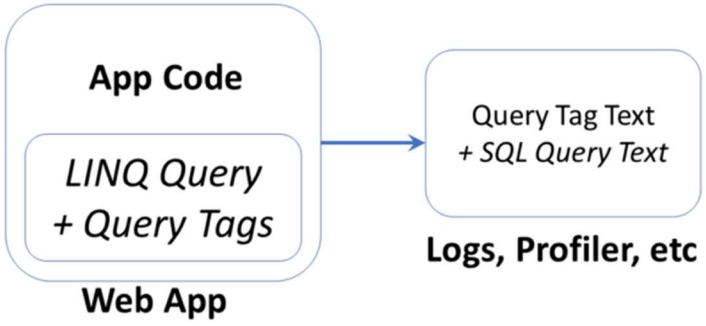
NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.17-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.17-alpha

In this Article:

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.



Query Tags in ASP .NET Core web apps

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

The NetLearner source code includes a C# model called LearningResource, with a few basic fields:

```
public class LearningResource
{
    public int Id { get; set; }

    [DisplayName("Resource")]
    public string Name { get; set; }


    [DisplayName("URL")]
    [DataType(DataType.Url)]
    public string Url { get; set; }

    public int ResourceListId { get; set; }
    [DisplayName("In List")]
    public ResourceList ResourceList { get; set; }

    [DisplayName("Feed Url")]
    public string ContentFeedUrl { get; set; }

    public List<LearningResourceTopicTag> LearningResourceTopicTags {
get; set; }
}
```

A collection of **LearningResource** objects are defined as a **DbSet** in the LibDbContext database context class:

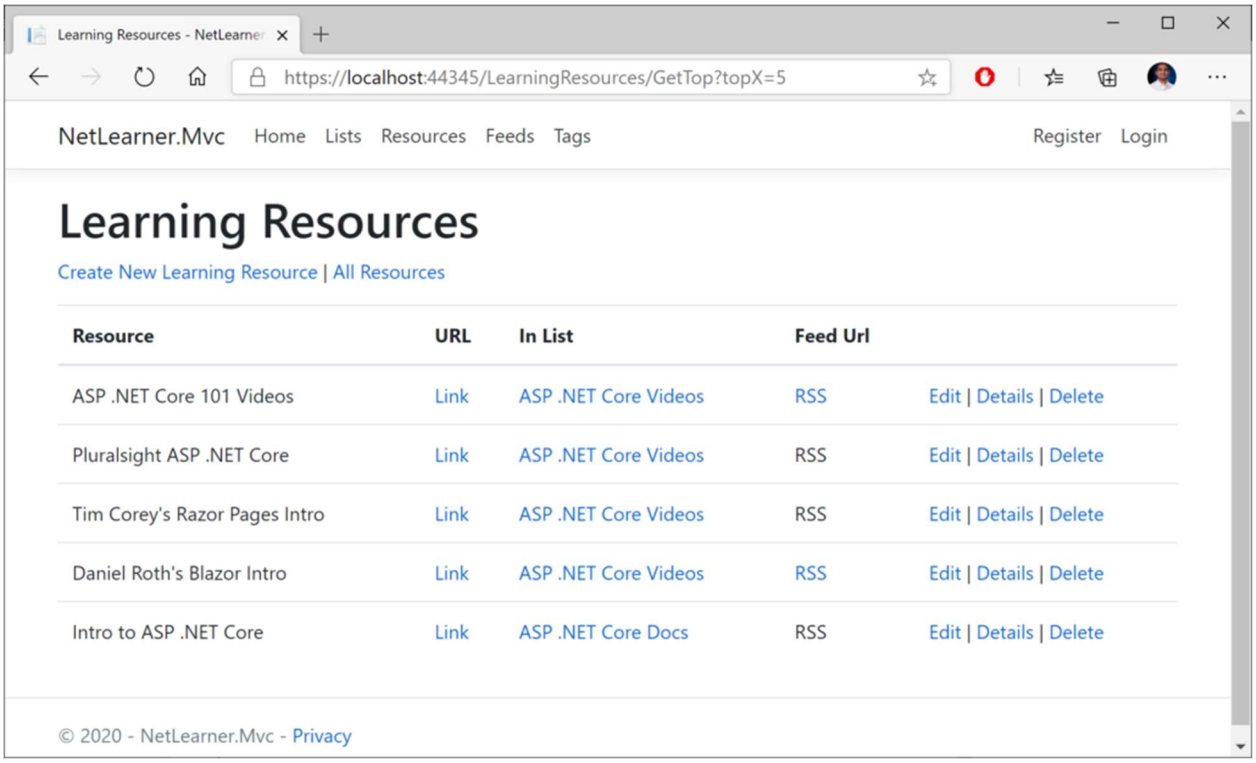
```
public DbSet<LearningResource> LearningResources { get; set; }
```

The **GetTop**() service method in the LearningResourceService class defines a Query Tag with the **TagWith**() method, as shown below:

```
public async Task<List<LearningResource>> GetTop(int topX)
{
    var myItems =
    (from m in _context.LearningResources
        .Include(r => r.ResourceList)
        .TagWith($"This retrieves top {topX} Items!")
        orderby m.Id ascending
        select m)
    .Take(topX);

    return (await myItems.ToListAsync());
}
```

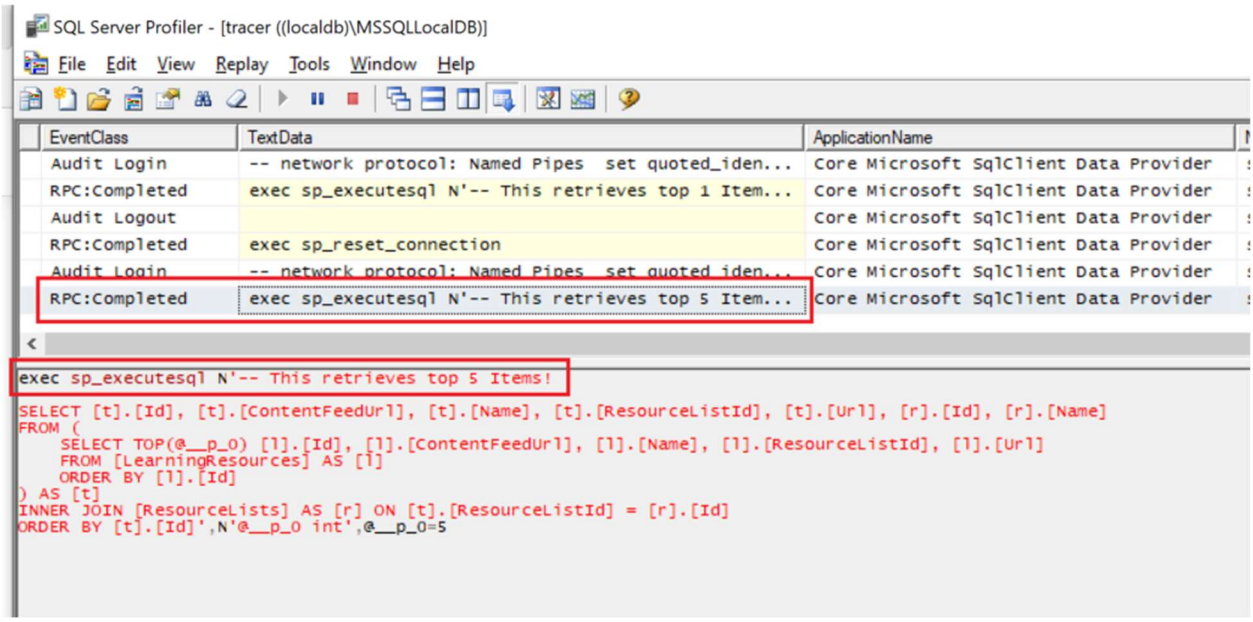
In the above query, the **TagWith**() method takes a single string value that can they be stored along with wherever the resultant SQL Queries are logged. This may include your persistent SQL Server database logs or Profiler logs that can be observed in real-time. It doesn't affect what gets displayed in your browser.

This can be triggered by visiting the LearningResourcesController's GetTop method, while passing in an integer value for topX. This causes the Index view to return the top "X" learning resources in a list. For example: if topX is set to 5, the resulting view will display 5 items in the displayed list.



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 5 in this example, the final string includes the value of topX inline within the logged text (more on formatting later).



SQL Server Profiler showing text from Query Tag

From the code documentation, the **TagWith**() method "a*dds 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 your code, you could 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!"
```

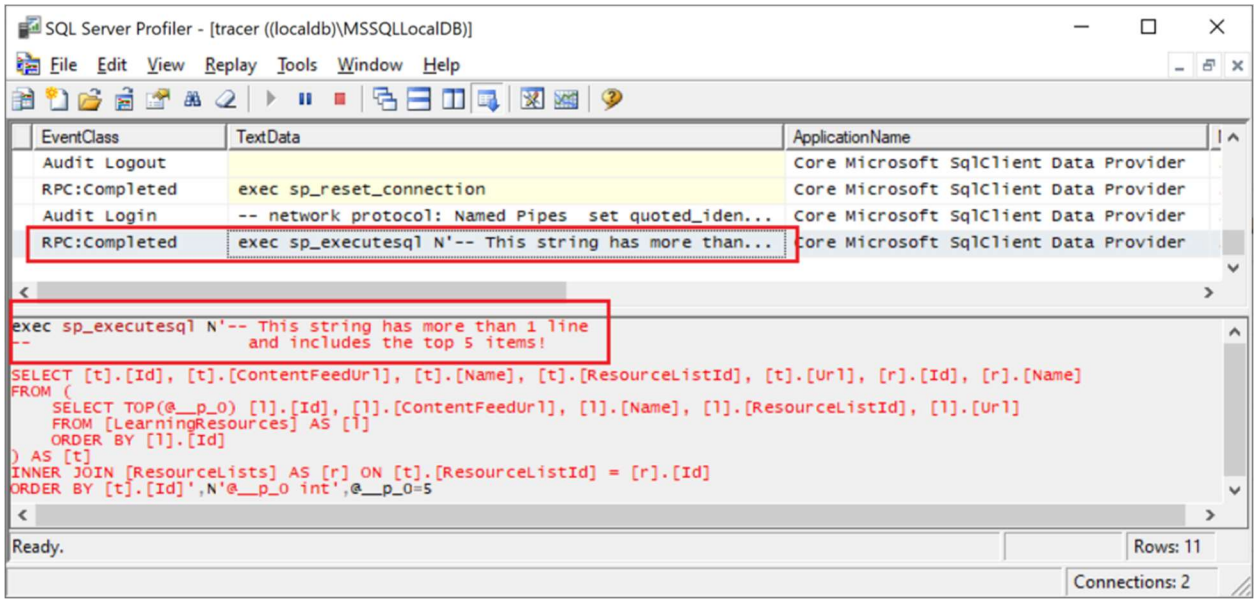
You may also have noticed that the profiler is showing the first comment in the same line as the leading text "**exec sp_executesql**" in the Profiler screenshot. If you want to add some better formatting (e.g. newline characters), you can use the so-called *verbatim identifier*, which is essentially 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} items!"
```

In an actual example, a newline produces the following results:



SQL Server Profiler, showing Query Tag text with newline

The above screenshot now shows multiline text from a Query Tag using newlines within the text string.

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 3.1

By Shahed C on May 4, 2020

3 Replies

This is the eighteenth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Query Tags in EF Core for ASP .NET Core 3.1 Web Apps

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.18-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.18-alpha

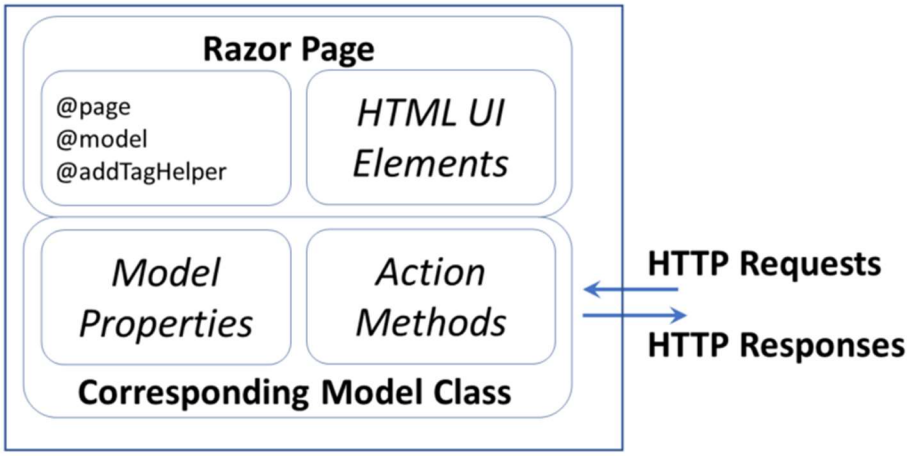
In this Article:

- R is for Razor Pages
- Core 3.x 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, and with more detail in my 2019 A-Z series. To complement this updated post in 2020, you may also refer to a previous posts in this series 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.



Razor Page with various elements, attributes and properties

Core 3.x Packages

To follow along, take a look at the NetLearner Razor Pages project on Github:

- NetLearner Razor Pages
  Project: https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Pages

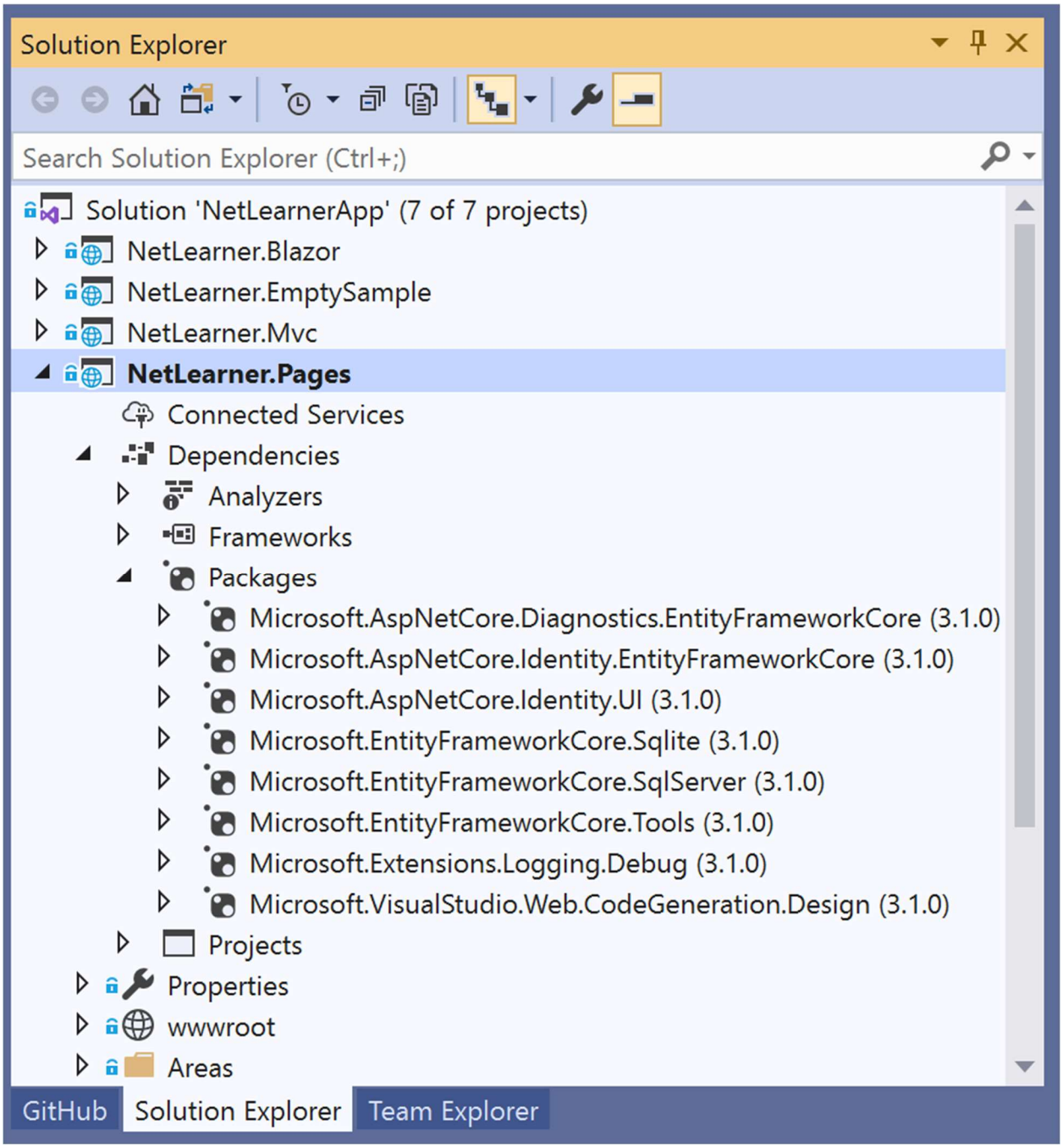
Let's start by taking a look at this 3.1 (LTS) project. The snippet below shows a .csproj for the sample app. This was created by starting with the Core 3.1 Razor Pages Template in VS2019 and then updating it to view/edit data from the shared NetLearner database.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <UserSecretsId>aspnet-NetLearner.Pages-38D1655D-B03D-469A-9E4E-
5DDDED554242</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <Compile Remove="Data\**" />
    <Content Remove="Data\**" />
    <EmbeddedResource Remove="Data\**" />
```

```
    <None Remove="Data\**" />
  </ItemGroup>


  <ItemGroup>
    <PackageReference
Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore"
Version="3.1.0" />
    <PackageReference
Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore"
Version="3.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.Identity.UI"
Version="3.1.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite"
Version="3.1.0" />
    <PackageReference
Include="Microsoft.EntityFrameworkCore.SqlServer" Version="3.1.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="3.1.0" />
    <PackageReference Include="Microsoft.Extensions.Logging.Debug"
Version="3.1.0" />
    <PackageReference
Include="Microsoft.VisualStudio.Web.CodeGeneration.Design"
Version="3.1.0" />
  </ItemGroup>


  <ItemGroup>
    <ProjectReference
Include="..\NetLearner.SharedLib\NetLearner.SharedLib.csproj" />
  </ItemGroup>

</Project>
```

For ASP .NET Core 3.x projects, 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 may show **NewtonsoftJson** and/or **EF Core** packages nested under the NuGet node, if you use them in your project.



Dependencies in Solution Explorer

If you need a refresher on the new changes for ASP .NET Core 3.x, 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

- [From 2018 A-Z 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/#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:

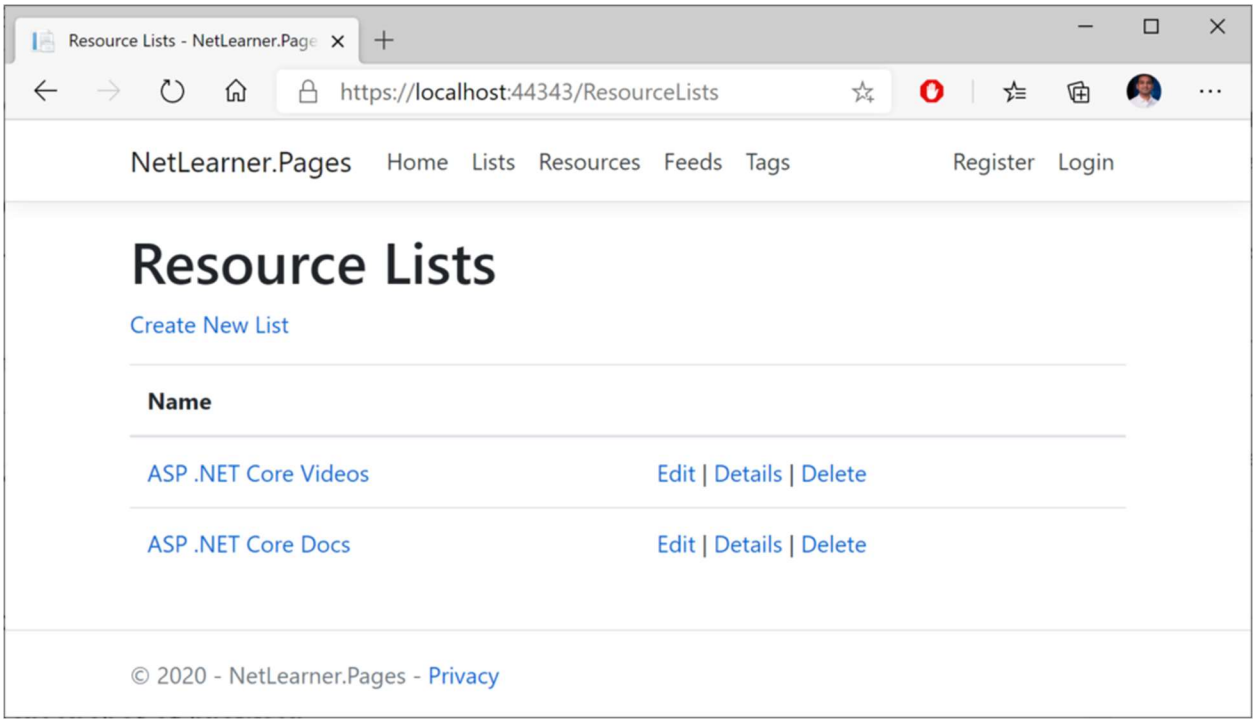
```
@page
@model IndexModel
@{
 ViewData["Title"] = "Home page";
}

<!-- 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 Razor Pages web project from NetLearner shows a list of Learning Resources at the following URL:

e.g. https://localhost:44343/ResourceLists



NetLearner's ResourceLists page

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 ResourceList ResourceList { 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, e.g.

```
[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 ResourceList ResourceList { get; set; }
```

Within the complex object, e.g. the shared ResourceList class has a public Id property:

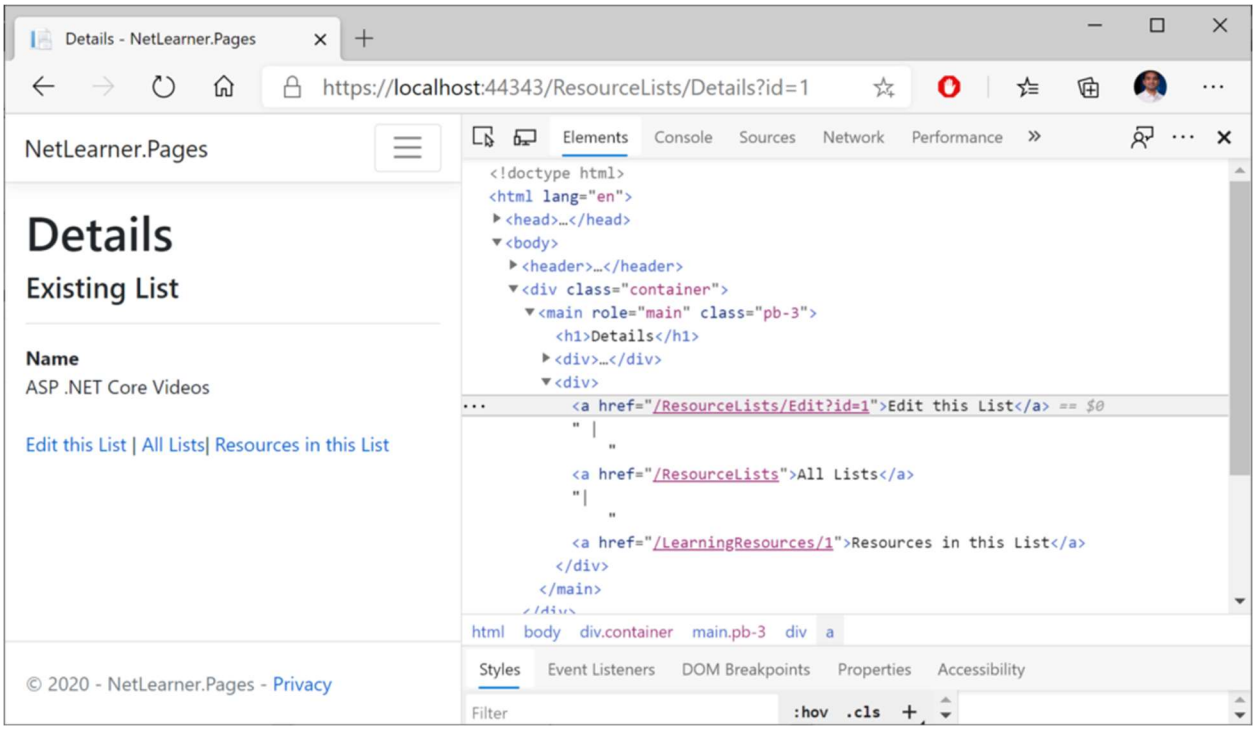
```
public int Id { get; set; }
```

In the Razor Page that refers to the above model, you can refer to **Model**.**Movie.ID** by name:

```
@page
@model NetLearner.Pages.DetailsModel
...
<a asp-page="./Edit" asp-route-id="@Model.ResourceList.Id">Edit this
List</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. "ResourceLists"), indicated by the period and slash in the path. The generated HTML looks like the following:

```
<a href="/ResourceLists/Edit?id=1">Edit this List</a>
```



Details page in browser with HTML source

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 ResourceLists 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" />
```

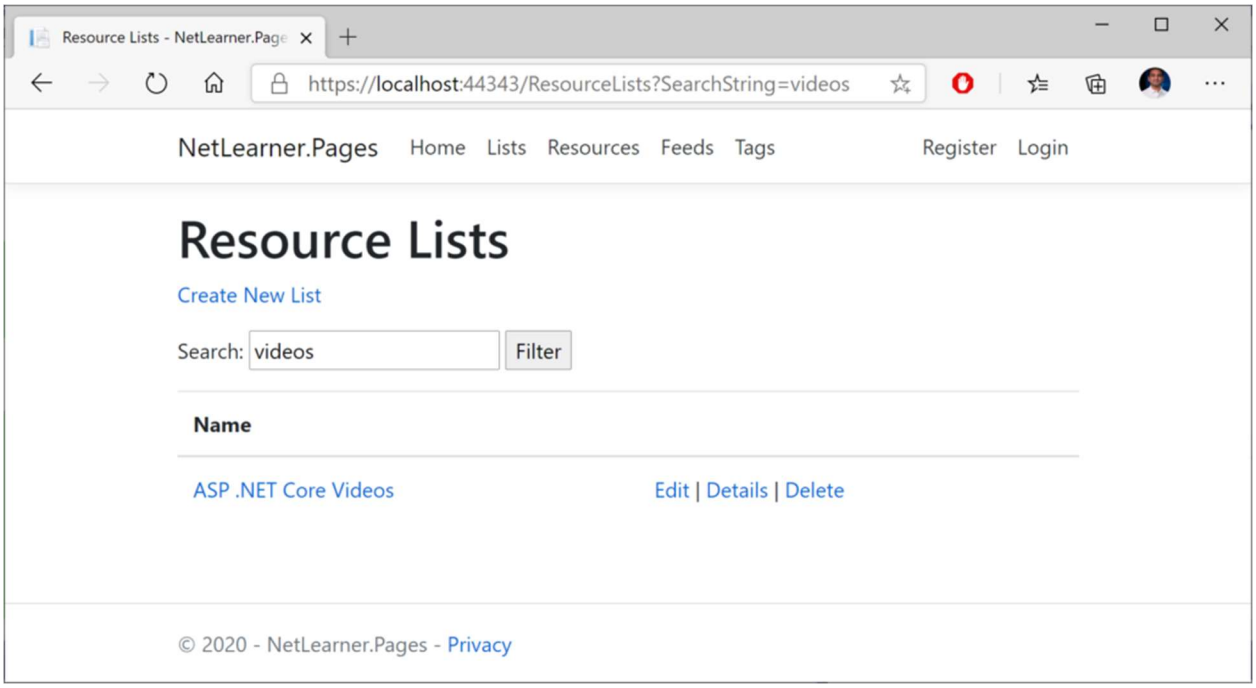
Page Routing

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 ResourceLists page with the search string “videos” includes any search results that include the term “videos”, as shown in the following screenshot.

e.g. https://localhost:44343/ResourceLists?SearchString=videos



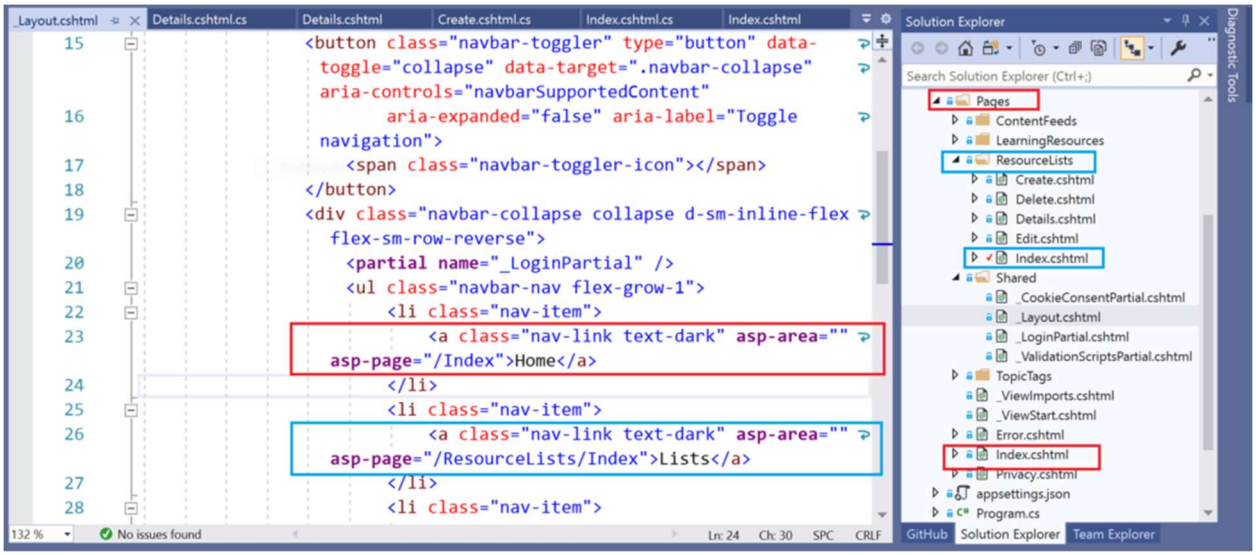
Index page for Resource Lists with ?SearchString parameter

Here, the value for **SearchString** is used by the **OnGetAsync**() method in the Index.cshtml.cs class for ResourceLists. 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<ResourceList> ResourceList { get;set; }
...
public async Task OnGetAsync()
{
    ...
    if (!string.IsNullOrEmpty(SearchString))
    {
       resourceLists = resourceLists.Where(s =>
s.Name.Contains(SearchString));
    }
    ...
    ResourceList = await resourceLists.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. “/ResourceLists/Index” as seen in the shared _Layout.cshtml file).

```
<a class="nav-link text-dark" asp-area="" asp-page="/Index">Home</a>

<a class="nav-link text-dark" asp-area="" asp-
page="/ResourceLists/Index">Lists</a>
```



Navigation links in Razor Pages, from Layout page

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.learnrazorpages.com/
- Razor Pages Syntax: https://www.learnrazorpages.com/razor-syntax
- Getting Started with Razor Pages: https://visualstudiomagazine.com/articles/2019/02/01/getting-started-with-razor.aspx

SignalR in ASP .NET Core 3.1

By Shahed C on May 11, 2020

6 Replies

This is the nineteenth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Razor Pages in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.19-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.19-alpha

In this Article:

S is for SignalR

SignalR has been around for 7+ years now, allowing ASP .NET developers to easily include real-time features in their web applications. SignalR Core has been available in ASP .NET Core since v2.1, as a cross-platform solution to add real-time features to web apps and more!

In this article, we'll go over SignalR concepts, using a new sample I developed to allow web users to vote in a real-time online poll. Before you begin, take a look at the sample code project on GitHub:

- SignalR Poll project on GitHub: https://github.com/shahedc/NetLearnerApp/tree/main/experimental/NetLearner.SignalRPoll

Back in 2018, I ran a couple of polls on Facebook and Twitter to see what the dev community wanted to see. On Twitter, the #1 choice was "Polling/Voting app" followed by "Planning Poker App" and "Real-time game". On Facebook, the #1 choice was "Real-time game" followed by "Polling/voting app". As a result, I decided to complement this article with a polling sample app.

- Twitter poll results: https://twitter.com/shahedC/status/1074862352787492864

POLL: What kind of #AspNetCore #SignalR apps would you like to see with https://t.co/lbmtwAyC6P Core 2.x? #webdev #webapps #realtime

— Shahed Chowdhuri @ Microsoft (@shahedC) December 18, 2018

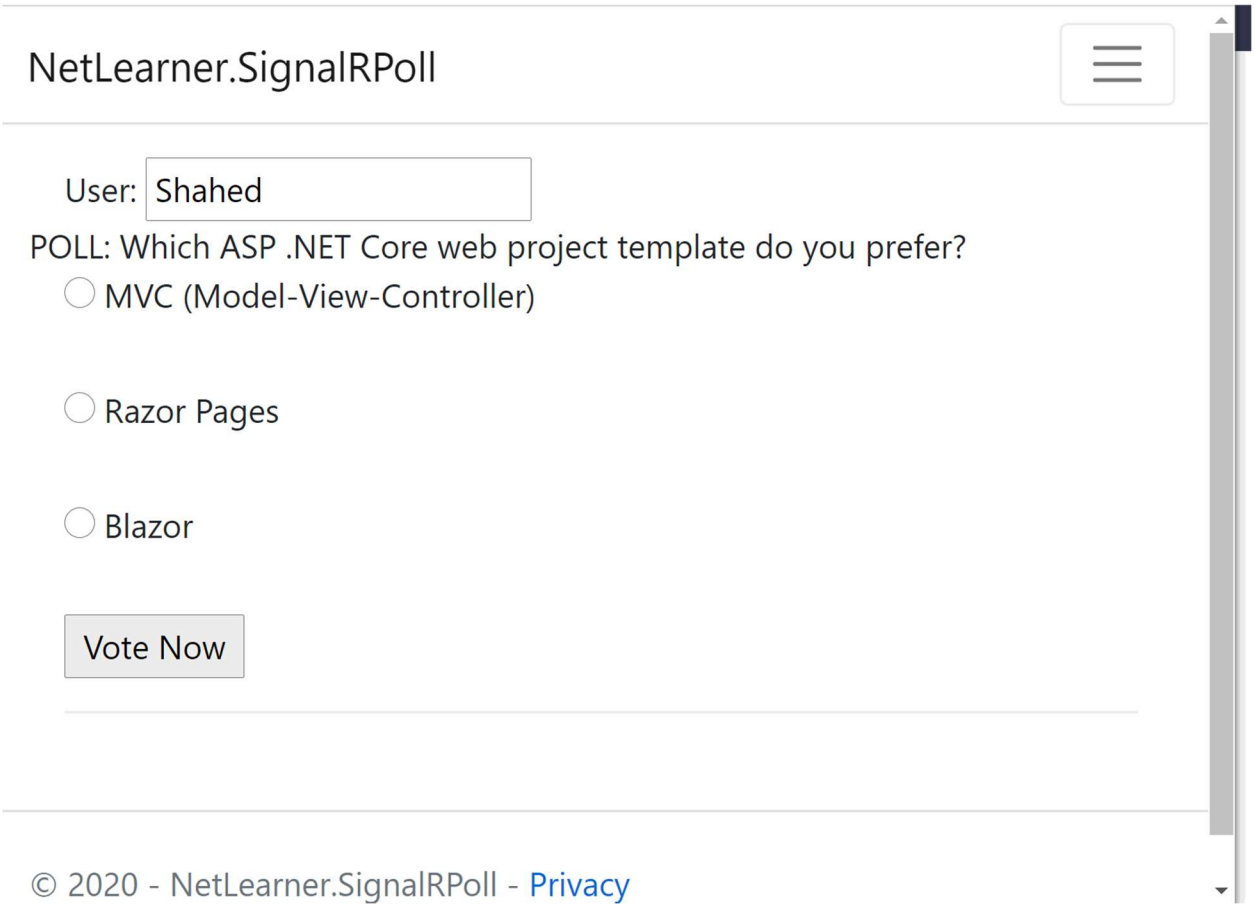
More importantly, Brady Gaster suggested that the sample app should definitely be "Not. Chat." 🙂

- "Not. Chat.": https://twitter.com/bradygaster/status/1075060352683933696

Not. Chat.

— Brady Gaster (@bradygaster) December 18, 2018

In the sample project, take a look at the **SignalRPoll** project to see how the polling feature has been implemented. In order to create a project from scratch, you'll be using both server-side and client-side dependencies.

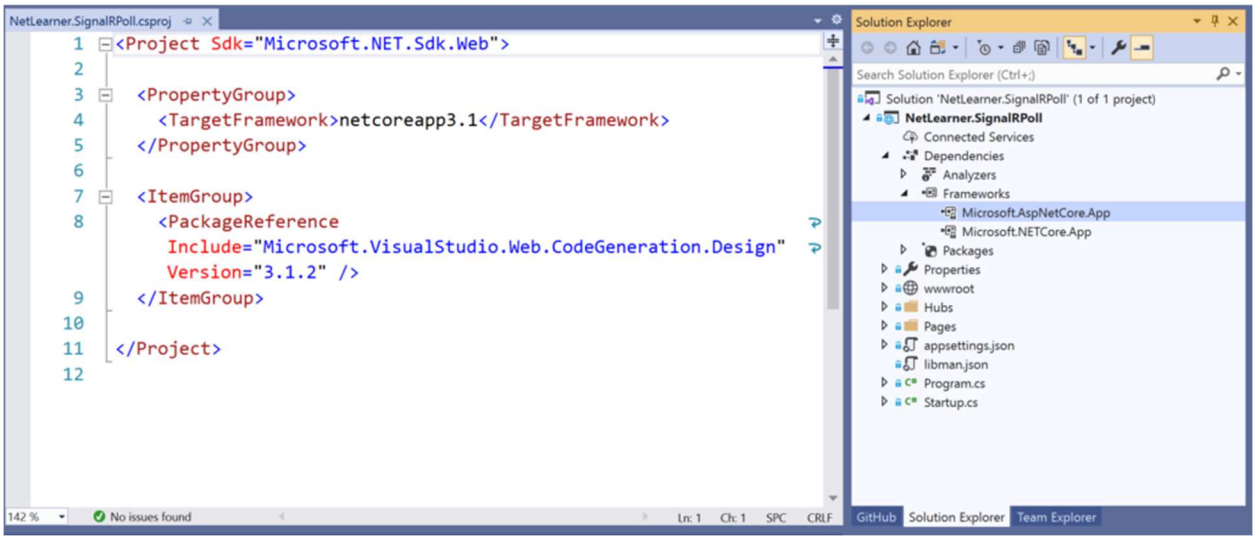


SignalR poll in action

If you need a starter tutorial, check out the official docs:

- Get started with ASP.NET Core SignalR: https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr

Dependencies



Visual Studio showing csproj + dependencies

The **Server-Side dependencies** for SignalR Core are available via the Microsoft.AspNetCore.App package so this is a freebie when you create a new 3.1 web app project. In your server-side code, you can use the following namespace:

```
using Microsoft.AspNetCore.SignalR;
```

This will give you access to SignalR classes such as Hub and Hub<T> for your SignalR hub to derive from. In the sample project, the PollHub class inherits from the Hub class. Hub<T> can be used for strongly-typed SignalR hubs.

The **Client Side dependencies** for SignalR Core have to be added manually. Simply right-click on your web app project and select Add | Client-Side Library. In the popup that appears, select a provider (such as “unpkg”) and enter a partial search term for Library, so that you can ideally pick the latest stable version.

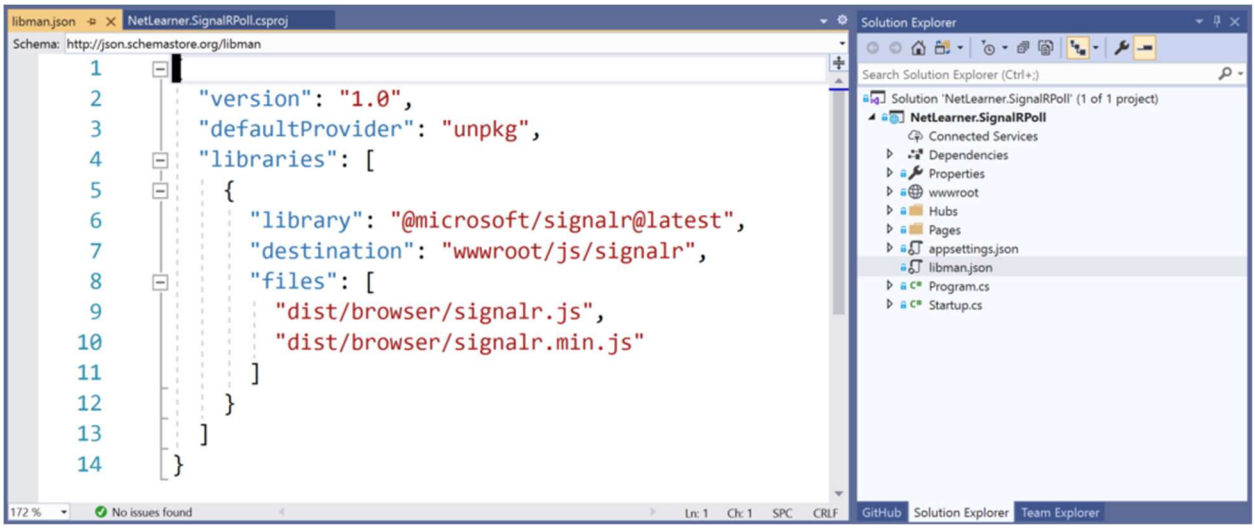
Steps to add client library via LibMan (aka Library Manager):

- 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: @aspnet/signalr@1… pick latest stable if desired
- **Files**: At a minimum, choose specific files signalr.js and/or its minified equivalent



Visual Studio, showing libman.json with client-side references

If you need help with adding client-side references, check out this earlier blog post in this A-Z series:

- JavaScript, CSS, HTML & Other Static Files in ASP .NET Core 3.1: https://wakeupandcode.com/javascript-css-html-other-static-files-in-asp-net-core-3-1/

Server-Side Hub

In the sample app, the PollHub class has a simple **SendMessage()** method with a few parameters. Derived from the sample Chat application, it starts with the user's desired "user" value and a custom "message" that can be passed to the SignalR Hub. For the the Captain Marvel/America poll, the method also passes an Id and Value for the selected radio button.

```
public class PollHub : Hub
{
   public async Task SendMessage(string user, string message, string
myProjectId, string myProjectVal)
   {
      await Clients.All.SendAsync("ReceiveMessage", user, message,
myProjectId, myProjectVal);
```

```
    }
}
```

To ensure that the SendMessage method from the server has a trigger on the client-side, the client-side code must invoke the method via the SignalR connection created with **HubConnectionBuilder**() on the client side. Once called, the above code will send a call to ReceiveMessage on all the clients connected to the Hub.

Client-Side

On the client-side, the JavaScript file poll.js handles the call from the browser to the server, and receives a response back from the server as well. The following code snippets highlight some important areas:

```
var connection = new
signalR.HubConnectionBuilder().withUrl("/pollHub").build();
...
connection.on("ReceiveMessage", function (user, message, myProjectId,
myProjectVal) {
    ...
    document.getElementById(myProjectId + 'Block').innerHTML +=
chartBlock;
});
...

document.getElementById("sendButton").addEventListener("click",
function (event) {
    ...
    connection.invoke("SendMessage", user, message, myProjectId,
myProjectVal)
    ...
});
```

The above snippets takes care of the following:

1. Creates a new connection objection using HubConnectionBuilder with a designated route
2. Uses connection.on to ensure that calls to ReceiveMessage come back from the server
3. Sets the innerHTML of a block to simulate a growing bar chart built with small blocks
4. Listens for a click event from the sendButton element on the browser
5. When the sendButton is clicked, uses connection.invoke() to call SendMessage on the server

Configuration

The configuration for the SignalR application is set up in the Startup.cs methods ConfigureServices() and Configure(), as you may expect.

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

public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
{
   ...
   app.UseEndpoints(endpoints =>
   {
      ...
      endpoints.MapHub<PollHub>("/pollHub");
   });
   ...
}
```

The above code takes care of the following:

1. the **ConfigureServices**() method adds SignalR to the ASP.NET Core dependency injection system with a call to **AddSignalR**()
2. the **Configure**() method adds SignalR to the middleware pipeline, while setting up the necessary route(s), using a call to **UseSignalR**().

At the time of this writing, I have more than one route set up for multiple hubs. For the polling app, we only need the call to **MapHub**<**PollHub**>() that sets up the route "/**pollHub**". You may recall this route from the client-side JavaScript code where the initial connection is set up.

For streaming fragments of data over time, you should also take a look at Streaming in SignalR Core:

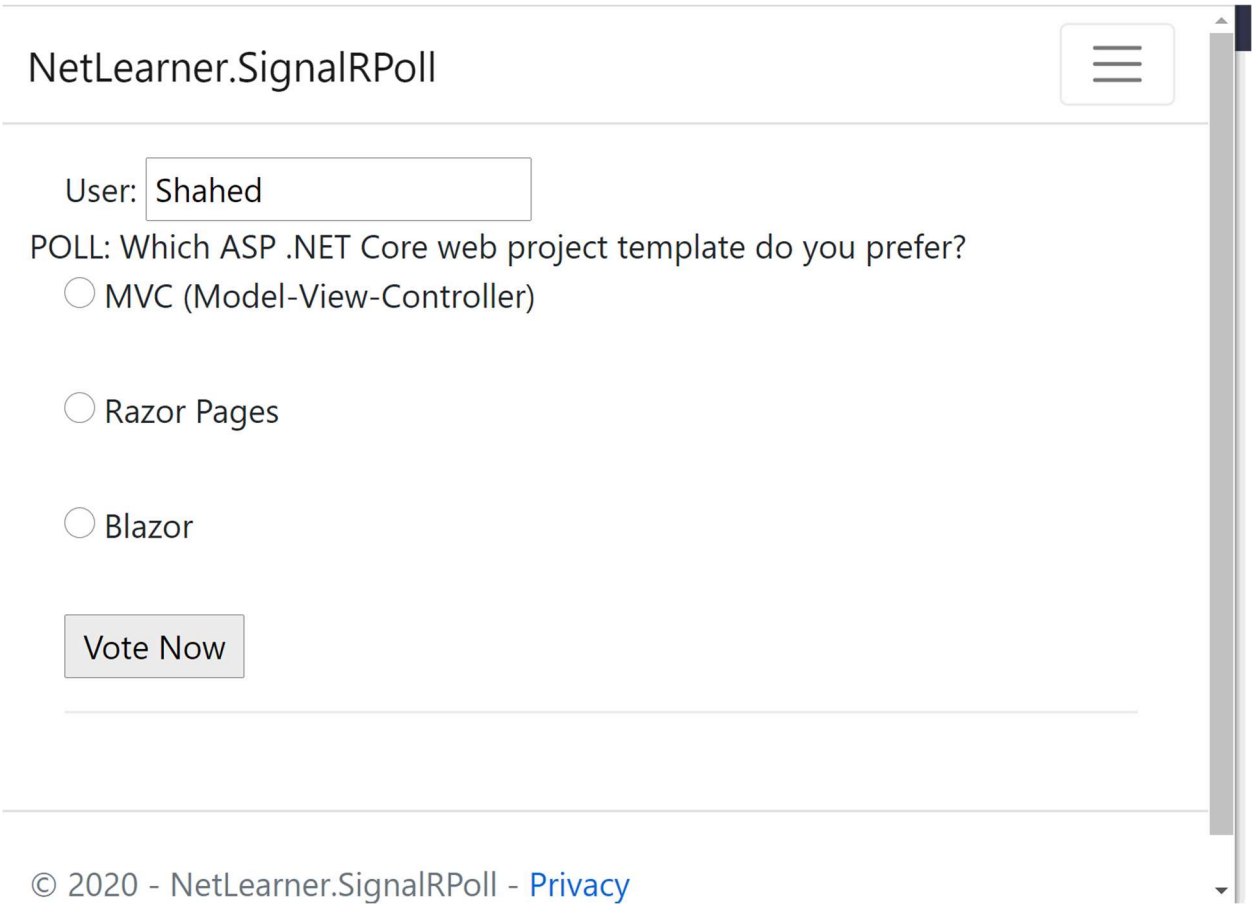
- Use streaming in ASP.NET Core SignalR: https://docs.microsoft.com/en-us/aspnet/core/signalr/streaming

Running the App

To run the app, simply run the SignalRPoll app Visual Studio or from the command line. Then, click the Poll item in the top menu to go to the Poll page. This page is a simple Razor page that contains all the HTML elements necessary to display the poll. It also includes <script> references to jQuery, SignalR and poll.js client-side references.

NOTE: Even though I am using jQuery for this sample, please note that jQuery is not required to use SignalR Core. On a related note, you can also configure Webpack and TypeScript for a TypeScript client if you want.

This GIF animation below illustrates the poll in action. To record this GIF of 1 browser window, I also launched additional browser windows (not shown) pointing to the same URL, so that I could vote several times.



SignalR poll in action

In a real world scenario, there are various ways to prevent a user from voting multiple times. Some suggestions include:

- Disable the voting button as soon as the user has submitted a vote.
- Use a cookie to prevent the user from voting after reloading the page.
- Use authentication to prevent a user from voting after clearing cookies or using a different browser.

For more information on authenticating and authorizing users, check out the official docs:

- Authentication and authorization in ASP.NET Core SignalR: https://docs.microsoft.com/en-us/aspnet/core/signalr/authn-and-authz

Azure SignalR Service

Azure SignalR Service is a fully-managed service available in Microsoft's cloud-hosted Azure services, that allows you to add real-time functionality and easily scale your apps as needed. Using Azure SignalR Service is as easy as 1-2-3:

1. Add a reference to the Azure SignalR Service SDK
2. Configure a connection string
3. Call services.AddSignalR().AddAzureSignalR() and app.UseAzureSignalR in Startup.cs

For more information on Azure SignalR Service, check out the official docs and tutorials:

- What is Azure SignalR: https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-overview
- C# Quickstart: https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-quickstart-dotnet-core

Packaging Changes in 3.x

You may have heard that ASP .NET Core 3.0 changed the way packages are made available to developers. So how does this affect SignalR for 3.x projects? Here is a recap from the official announcement:

- Microsoft "will stop producing many of the NuGet packages that we have been shipping since ASP.NET Core 1.0. The API those packages provide are still available to apps by using a <FrameworkReference> to Microsoft.AspNetCore.App. This includes commonly referenced API, such as Kestrel, Mvc, Razor, and others."
- "This will not apply to all binaries that are pulled in via Microsoft.AspNetCore.App in 2.x."

- "***Notable exceptions include***: The SignalR .NET client will continue to support .NET Standard and ship as NuGet package because it is intended for use on many .NET runtimes, like Xamarin and UWP."

Source: https://github.com/aspnet/Announcements/issues/325

References:

- Intro to ASP.NET Core SignalR: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction
- Get started with ASP.NET Core SignalR: https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr
- Create backend services for native mobile apps with ASP.NET Core: https://docs.microsoft.com/en-us/aspnet/core/mobile/native-mobile-backend
- Use ASP.NET Core SignalR with TypeScript and Webpack: https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr-typescript-webpack
- SignalR Service C# Quickstart: https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-quickstart-dotnet-core

Tag Helper Authoring in ASP .NET Core 3.1

By Shahed C on May 18, 2020

5 Replies

This is the twentieth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- SignalR in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.20-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.20-alpha

NOTE: The NetLearner suite of apps doesn't currently use custom tag helpers in the main branch, so you can check out the new sample code in the experimental subfolder, merged from a branch:

- Tag Helper Authoring: https://github.com/shahedc/NetLearnerApp/tree/TagHelpers/experimental/TagHelperAuthoring

In this Article:

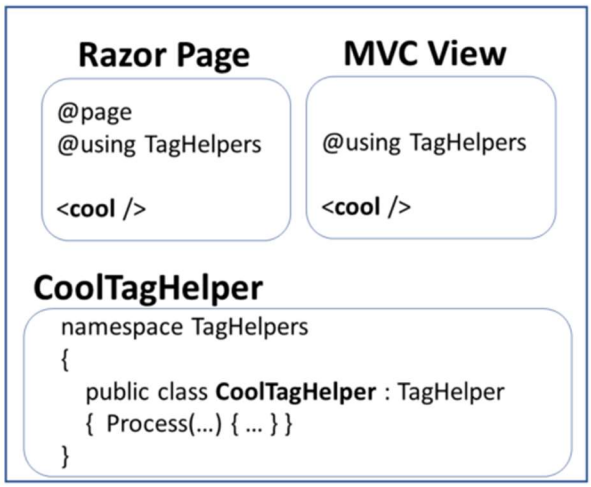
- Setting Attributes and Content
- Updating Pre/Post Content
- Passing Complex Objects
- Handling Conditions
- References

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-3-1/



Tag Helpers 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.

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 was originally written to cover 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.

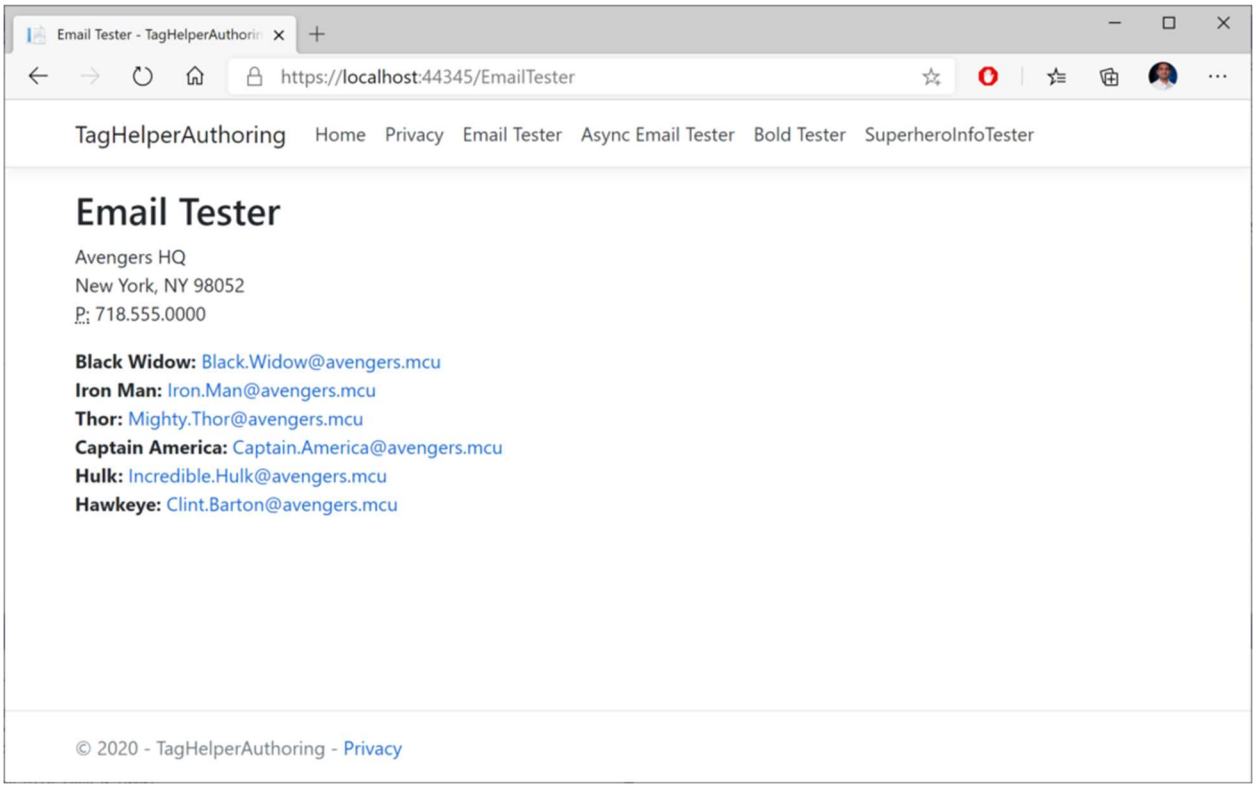
In the EmailTester.cshtml page:

```
<email mail-to="Black.Widow"></email>
```

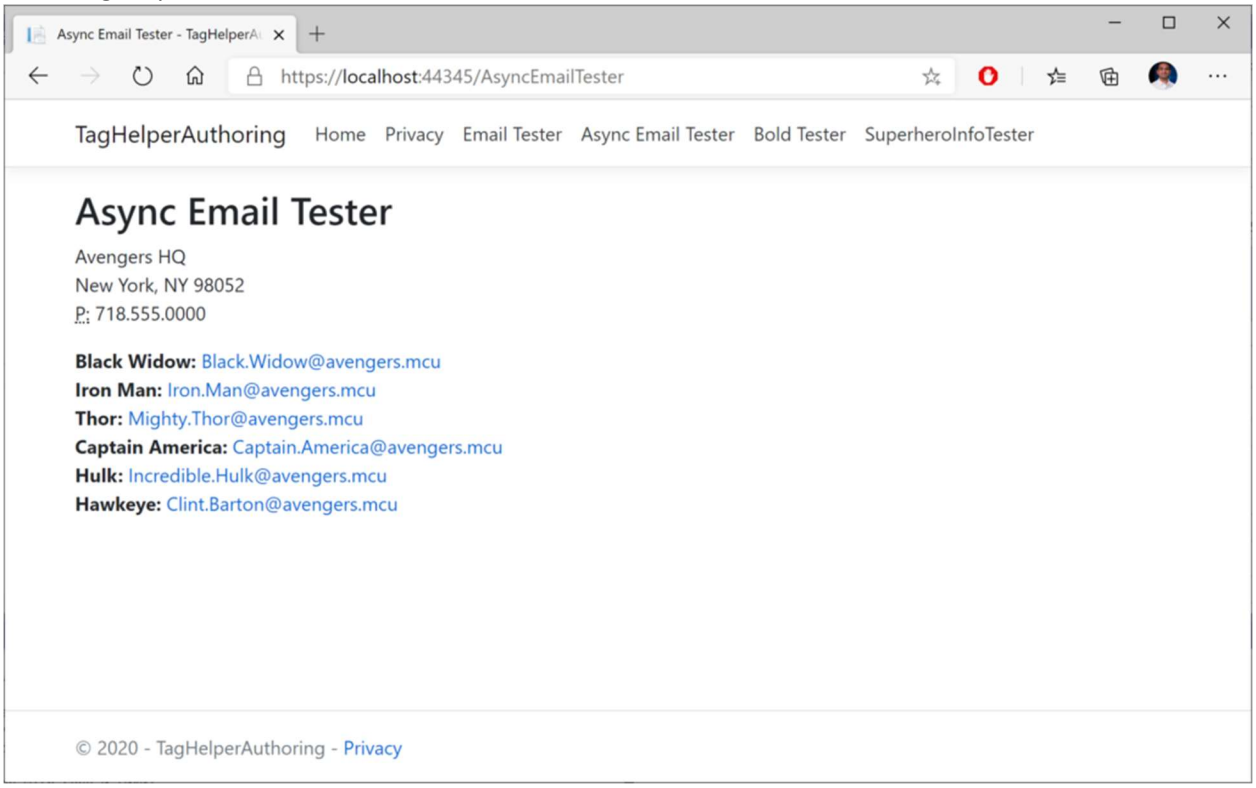
In the AsyncEmailTester.cshtml page:

```
<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 tag helper in a browser



Async Email tag helper in a browser

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>");
   }
}
```

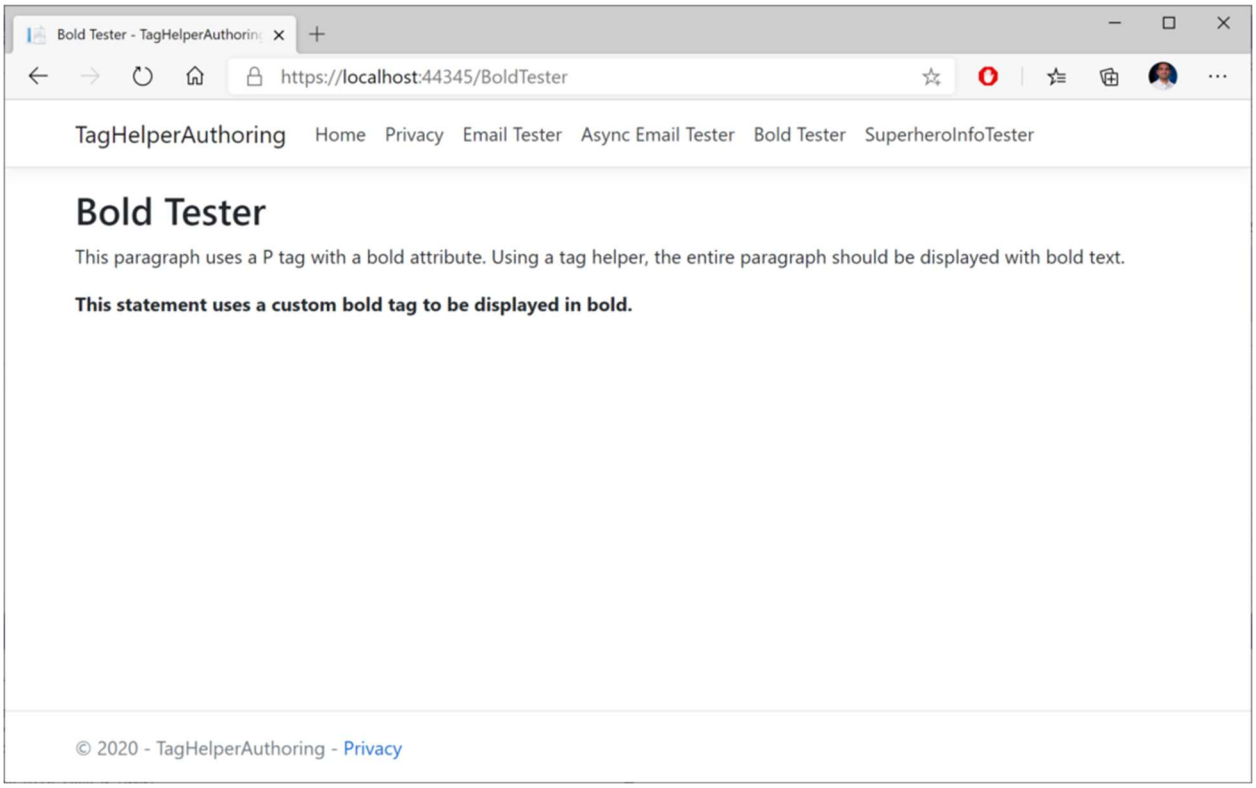
Let's go over what the code does, line by line:

- 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 **bold** *attribute* 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:



Bold tag helper in a browser

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 or <bold>
- The **SetHtmlContent()** from **output.Postontent** adds a *closing* </**strong**> tag insidethe enclosing element, i.e. just before 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 (SuperheroInfoTester.cshtml.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.

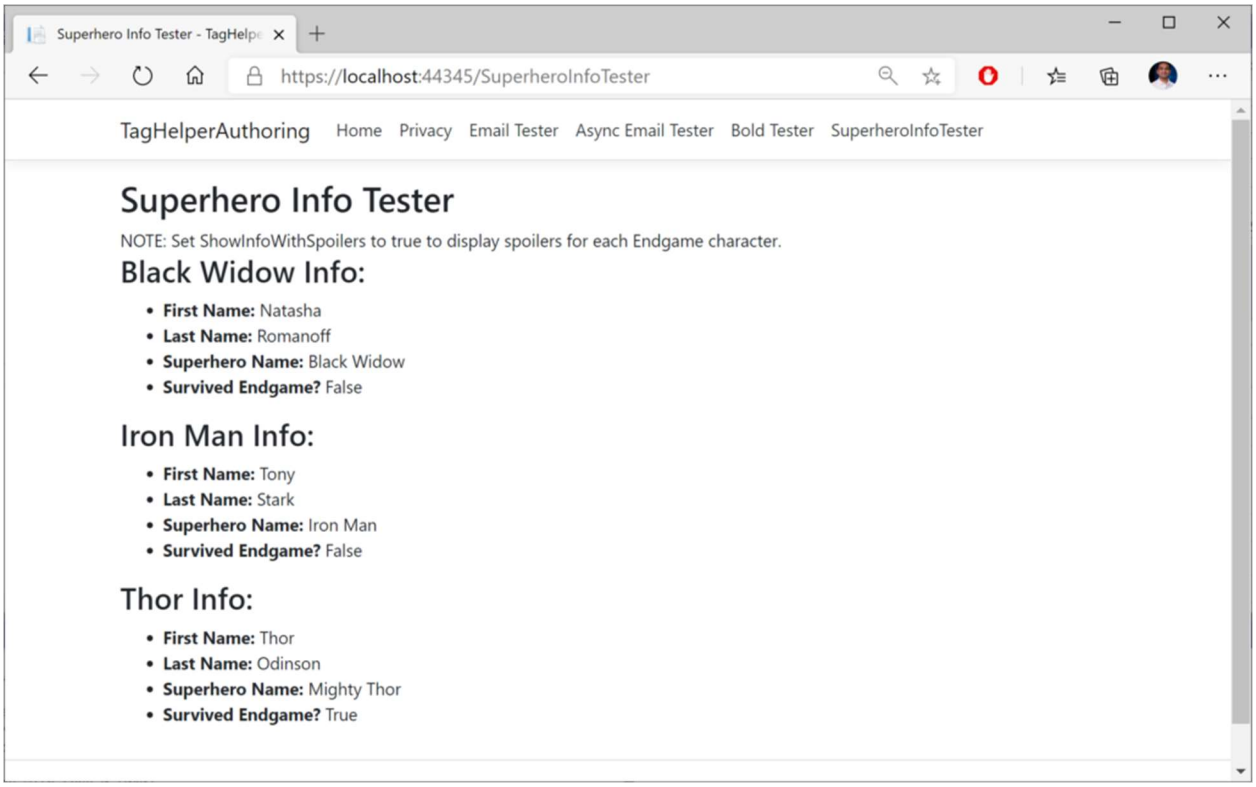
Inside the tag helper, the Process() method takes care of replacing the <**superhero**> tag with a <**section**> tag:

```
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:



Superhero tag helper in a browser

In a web browser, you can see that 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 3.1

By Shahed C on May 25, 2020

5 Replies

This is the twenty-first of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Tag Helpers in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.21-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.21-alpha

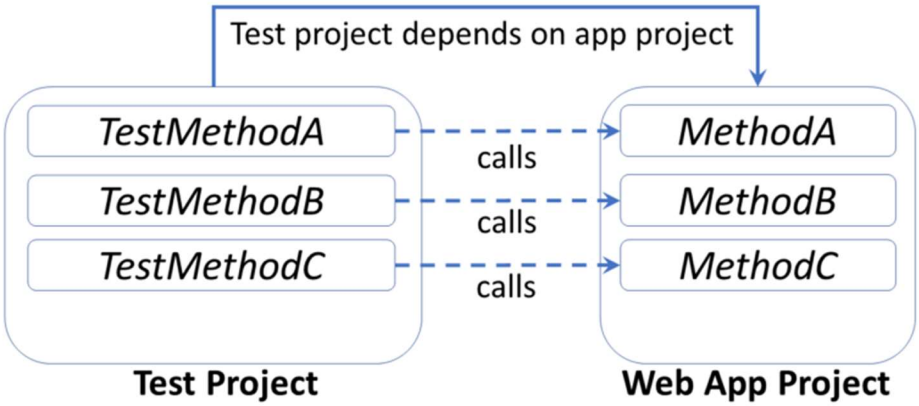
In this Article:

- 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.



Unit Testing Project Dependencies

Here are some poll results, from asking 500+ developers about which testing framework they prefer, showing xUnit.net in the lead (from May 2019).

- Twitter poll: https://twitter.com/shahedC/status/1131337874903896065

POLL: Hey #AspNetCore #webdev community on twitter! What do you use for unit testing #ASPNET web apps?

Please RT for reach. Thanks!

— Shahed Chowdhuri @ Microsoft (@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:

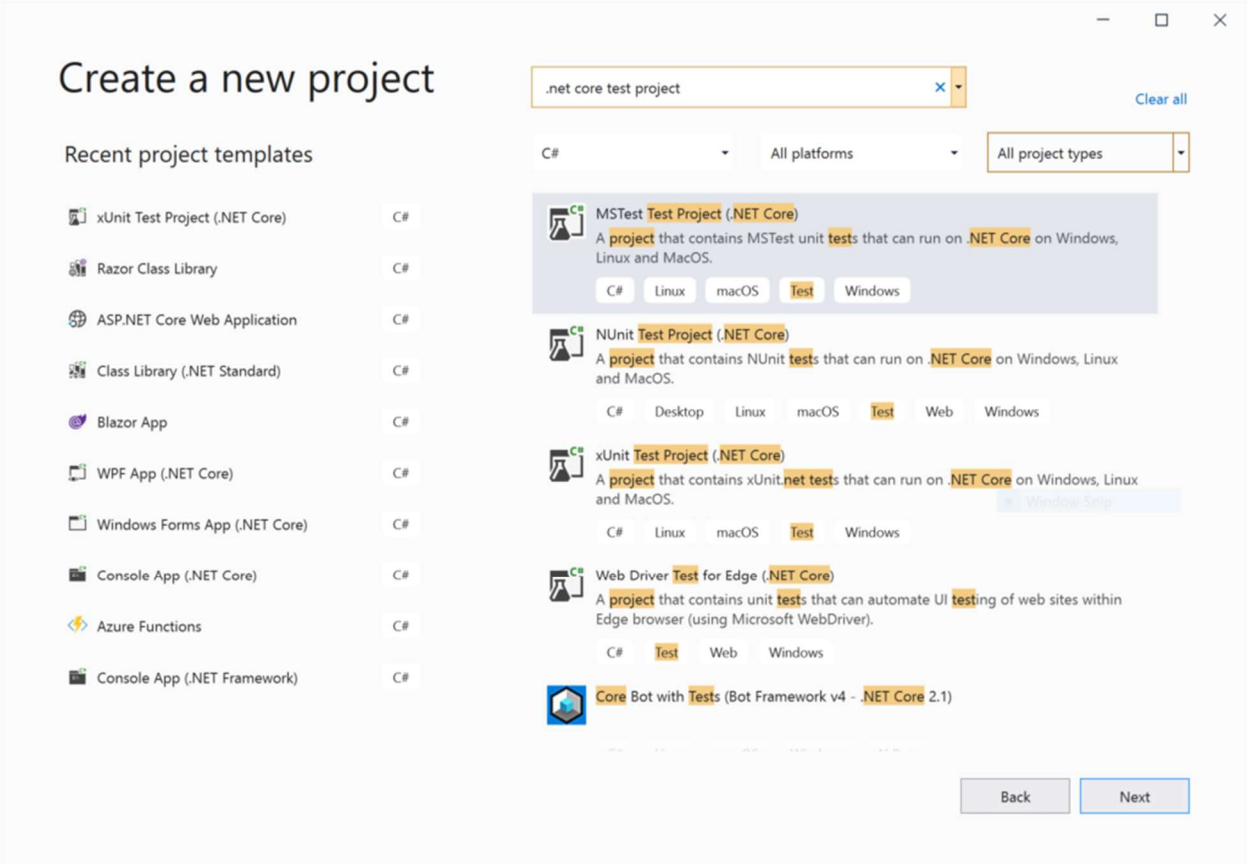
- Comparing xUnit.net to other frameworks: https://xunit.net/docs/comparisons

To follow along, take a look at the test projects on Github:

- Shared Library tests: https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.SharedLib.Tests
- MVC tests: https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Mvc.Tests
- Razor Pages tests: https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Pages.Tests
- Blazor tests: https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Blazor.Tests

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 NetLearner samples.



Test Project Templates in Visual Studio 2019

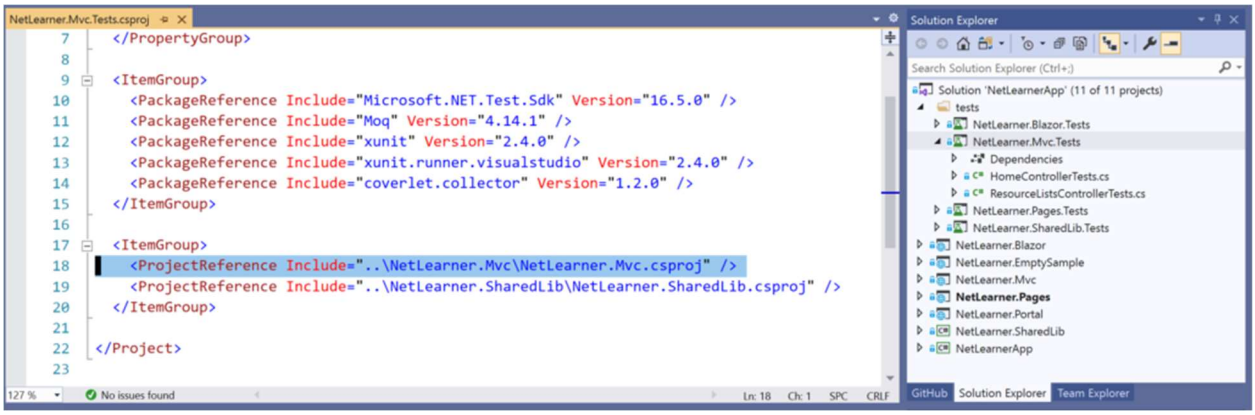
The placeholder unit test class includes a blank test. Typically, you could create a test class for each application class being tested. The simplest unit test usually includes three distinct steps: Arrange, Act and Assert.

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 NetLearner.Mvc.Tests.csproj, you'll find a reference to NetLearner.Mvc.csproj.

```
 ...
<ItemGroup>
   <ProjectReference Include="..\NetLearner.Mvc\NetLearner.Mvc.csproj"
/>
</ItemGroup>
...
```

In the Solution Explorer panel, you should see a project dependency of the reference project.



Project Reference in Unit Testing 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 NetLearner Shared Library test project, you'll see a test class (ResourceListServiceTests.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 of more [**InlineData**] attributes that have sample argument values for each method parameter.

```
[Theory(DisplayName = "Add New Resource List")]
[InlineData("RL1")]
public async void TestAdd(string expectedName)
{
   ...
}
```

In the code sample, each occurrence of [InlineData] should reflect the test method's parameters, e.g.

- **[InlineData**("RL1")] –> this implies that expectedName = "RL1"

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 **TestAdd**() method and its method body.

```
public async void TestAdd(string expectedName)
{
    var options = new DbContextOptionsBuilder<LibDbContext>()
        .UseInMemoryDatabase(databaseName: "TestNewListDb").Options;

    // Set up a context (connection to the "DB") for writing
    using (var context = new LibDbContext(options))
    {
        // 1. Arrange
        var rl = new ResourceList
        {
            Name = "RL1"
        };
```

```
        // 2. Act
        var rls = new ResourceListService(context);
        await rls.Add(rl);
    }

    using (var context = new LibDbContext(options))
    {
        var rls = new ResourceListService(context);
        var result = await rls.Get();

        // 3. Assert
        Assert.NotEmpty(result);
        Assert.Single(result);
        Assert.NotEmpty(result.First().Name);
        Assert.Equal(expectedName, result.First().Name);
    }
}
```

1. During the **Arrange** step, we create a new instance of an object called **ResourceList** which will be used during the test.
2. During the **Act** step, we create a **ResourceListService** object to be tested, and then call its **Add**() method to pass along a string value that was assigned via InlineData.
3. During the **Assert** step, we compare the **expectedName** (passed by InlineData) with the returned result (obtained from a call to the Get method in the service 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, e.g.

```
Exception ex = Assert
    .Throws<SpecificException>(() => someObject.MethodBeingTested(x,
y));
```

The above code tests a method named **MethodBeingTested**() for someObject being tested.
A **SpecificException**() is expected to occur when the parameter values x and y are passed in. In this case, the Act and Assert steps occur in the same statement.

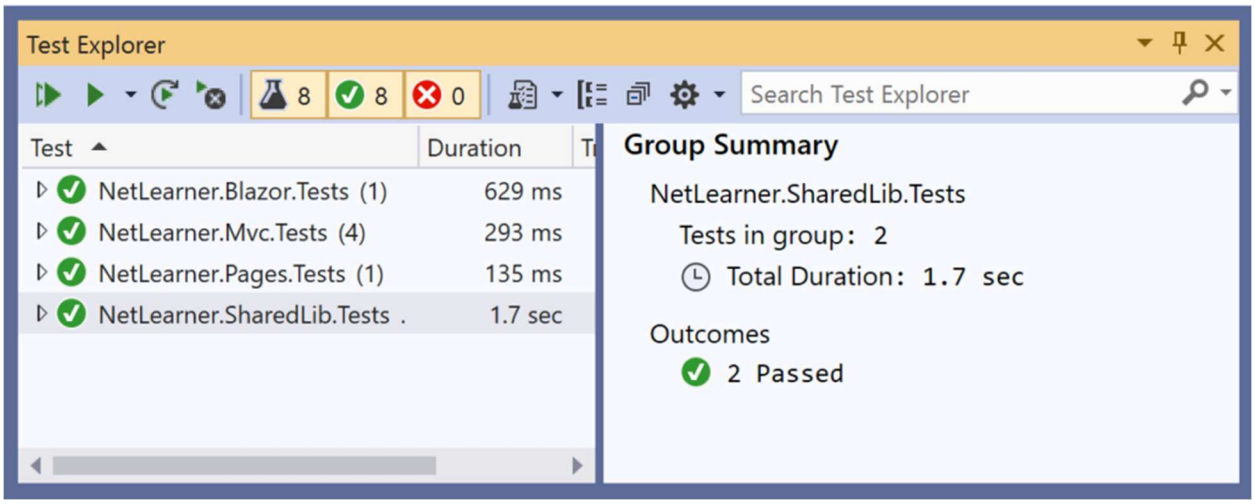
NOTE: There are some differences in opinion whether or not to use InMemoryDatabase for unit testing. Here are some viewpoints from .NET experts Julie Lerman (popular Pluralsight author) and Nate Barbettini (author of the Little ASP .NET Core book):

- Julie Lerman: *"If you use it wisely and are benefiting (not as a replacement for integration tests) then they are AOK in my mind."*
- Source: https://twitter.com/julielerman/status/1259939718151770112
- Nate Barbettini: *"I still might use InMemory for early rapid prototyping work, but no longer use it for testing. I don't let any unit tests talk to the DB (pure classes and methods under test only), and no longer use InMemory for integration tests."*
- Source: https://twitter.com/nbarbettini/status/1259939958573248514
- Discussion on GitHub: https://github.com/dotnet/efcore/issues/18457

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. If any tests fail, inspect the code and fix as needed.

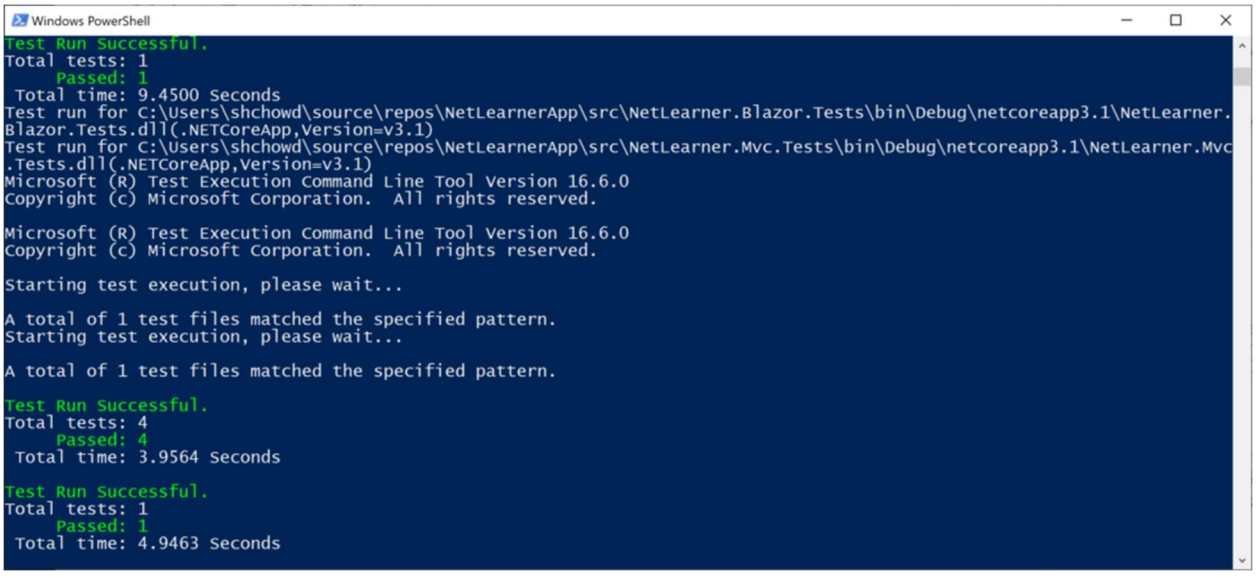


Test Explorer in VS2019

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:



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

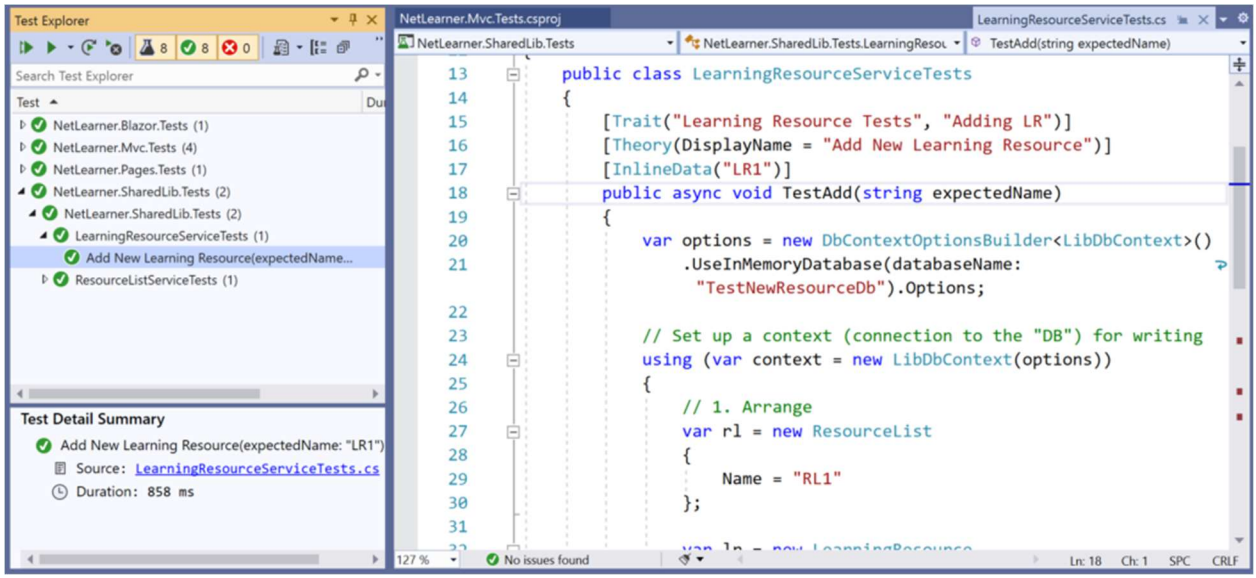
NOTE: Visual Studio includes a Live Unit Testing feature that allows you to see the status of passing/failing tests as you're typing your code. This feature is only available in the Enterprise Edition of Visual Studio.

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 defined a friendly name for any test method (Fact or Theory) that can be displayed in the Test Explorer. For example:

```
[Theory(DisplayName = "Add New Learning Resource")]
```

Using the above attribute above the **TestAdd**() method will show the friendly name "**Add New Learning Resource**" in the Test Explorer panel during test runs.

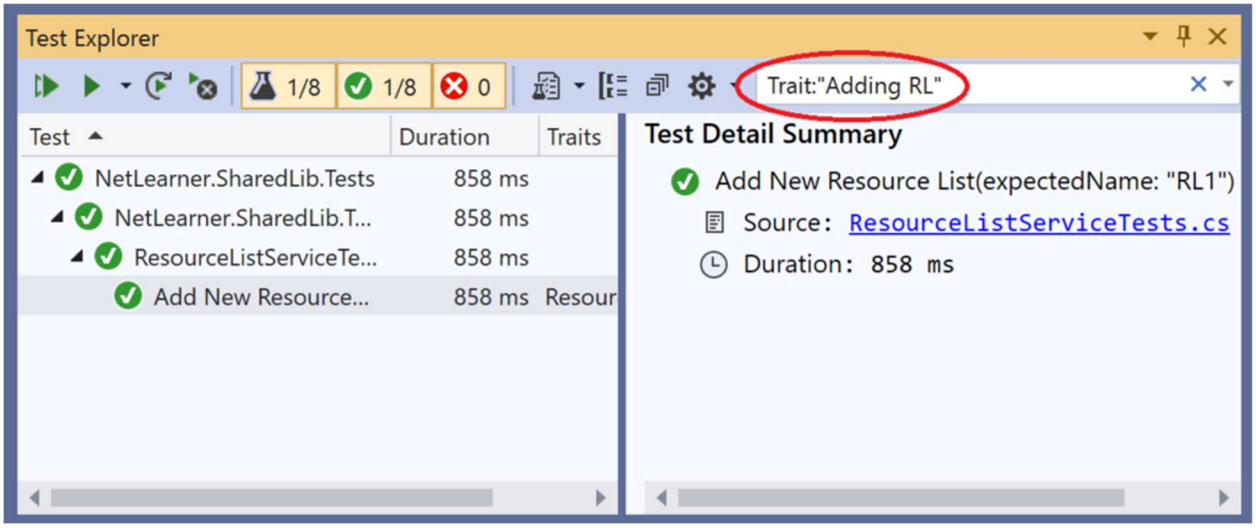


Unit Test with custom DisplayName

Finally, consider the [**Trait**] attribute. This attribute can be use to categorize related test methods by assigning an arbitrary name/value pair for each defined "Trait". For example (from LearningResource and ResourceList tests, respectively):

```
[Trait("Learning Resource Tests", "Adding LR")]
public void TestAdd() { ... }

[Trait("Resource List Tests", "Adding RL")]
public void TestAdd { ... }
```

Using the above attribute for the two **TestAdd**() methods will categorize the methods into their own named “category”, e.g. Learning Resource Tests and Resource List Tests. This makes it possible to filter just the test methods you want to see, e.g. Trait: “Adding RL”



Filtering Unit Tests by Trait Values

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 multiples 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 goes into the shared library, the same concepts carry over into the testing of each individual web app project (MVC, Razor Pages and Blazor). Refer to the following documentation and blog content for each:

- MVC unit tests in ASP .NET Core: https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/testing?view=aspnetcore-3.1
- Razor Pages unit tests in ASP.NET Core: https://docs.microsoft.com/en-us/aspnet/core/test/razor-pages-tests
- Blazor unit tests with bUnit (Preview): https://www.telerik.com/blogs/unit-testing-blazor-components-bunit-justmock

Refer to the NetLearner sample code for unit tests for each web project:

- MVC Controller: https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Mvc.Tests/ResourceListsControllerTests.cs
- Razor Pages: https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Pages.Tests/ResourceListPageTests.cs

- Blazor: https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Blazor.Tests/BlazorTest.cs

In order to set up a shared service object to be used by the controller/page/component being tested, Moq is used to mock the service. For more information on Moq, check out their official documentation on GitHub:

- Moq on GitHub: https://github.com/Moq/moq4/wiki/Quickstart

For the Blazor testing project, the following references were consulted:

- Telerik Blazor Testing Guide (using bUnit beta 6): https://www.telerik.com/blogs/unit-testing-blazor-components-bunit-justmock
- bUnit on GitHub (currently at beta 7): https://github.com/egil/bunit

NOTE: Due to differences between bUnit beta 6 and 7, there are some differences between the Blazor guide and the NetLearner tests on Blazor. I started off with the Blazor guide, but made some notable changes.

1. Instead of starting with a Razor Class Library template for the test project, I started with the xUnit Test Project template.
2. There was no need to change the test project's target framework from .NET Standard to .NET Core 3.1 manually, since the test project template was already Core 3.1 when created.
3. As per the bUnit guidelines, the test class should no longer be derived from the ComponentTestFixture class, which is now obsolete: https://github.com/egil/bunit/blob/6c66cc2c77bc8c25e7a2871de9517c2fbe6869dd/src/bunit.web/ComponentTestFixture.cs
4. Instead, the test class is now derived from the TestContext class, as seen in the bUnit source code: https://github.com/egil/bunit/blob/6c66cc2c77bc8c25e7a2871de9517c2fbe6869dd/src/bunit.web/TestContext.cs

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/
- Integration tests with default Web Application Factory: https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-3.1#basic-tests-with-the-default-webapplicationfactory
- 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/mocksArentStubs.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/
- Fake Data with Bogus: https://github.com/bchavez/Bogus

XML + JSON Output for Web APIs in ASP .NET Core 3.1

By Shahed C on June 22, 2020

3 Replies

This is the twenty-fourth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- Worker Service in .NET Core 3.1

NetLearner on GitHub:

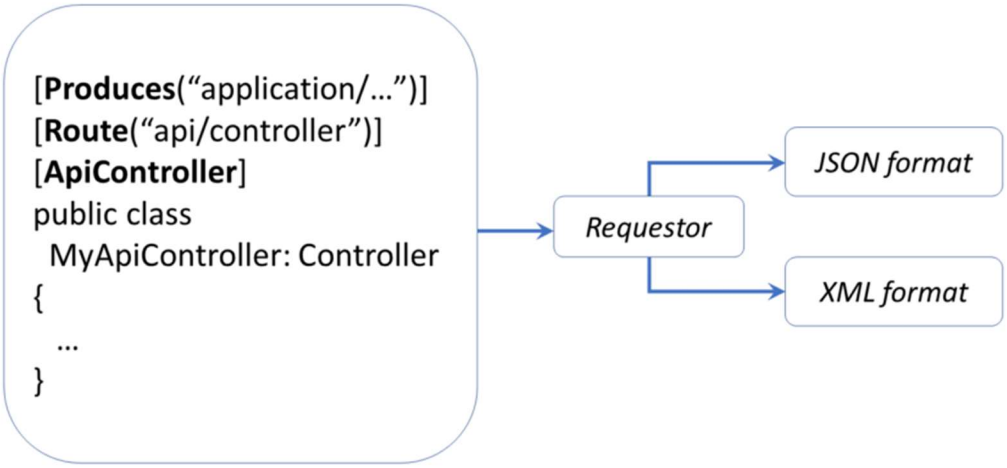
- Repository: https://github.com/shahedc/NetLearnerApp
- v0.24-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.24-alpha

In this Article:

- X is for XML + JSON Output
- Returning JsonResult and IActionResult
- Returning Complex Objects
- XML Output
- References

X is for XML + JSON Output

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 object literals of JavaScript, but has also been used for storing data in both structured and unstructured formats, regardless of the language used. In fact, ASP .NET Core applications switched from XML-based .config files to JSON-based .json settings files for application configuration.



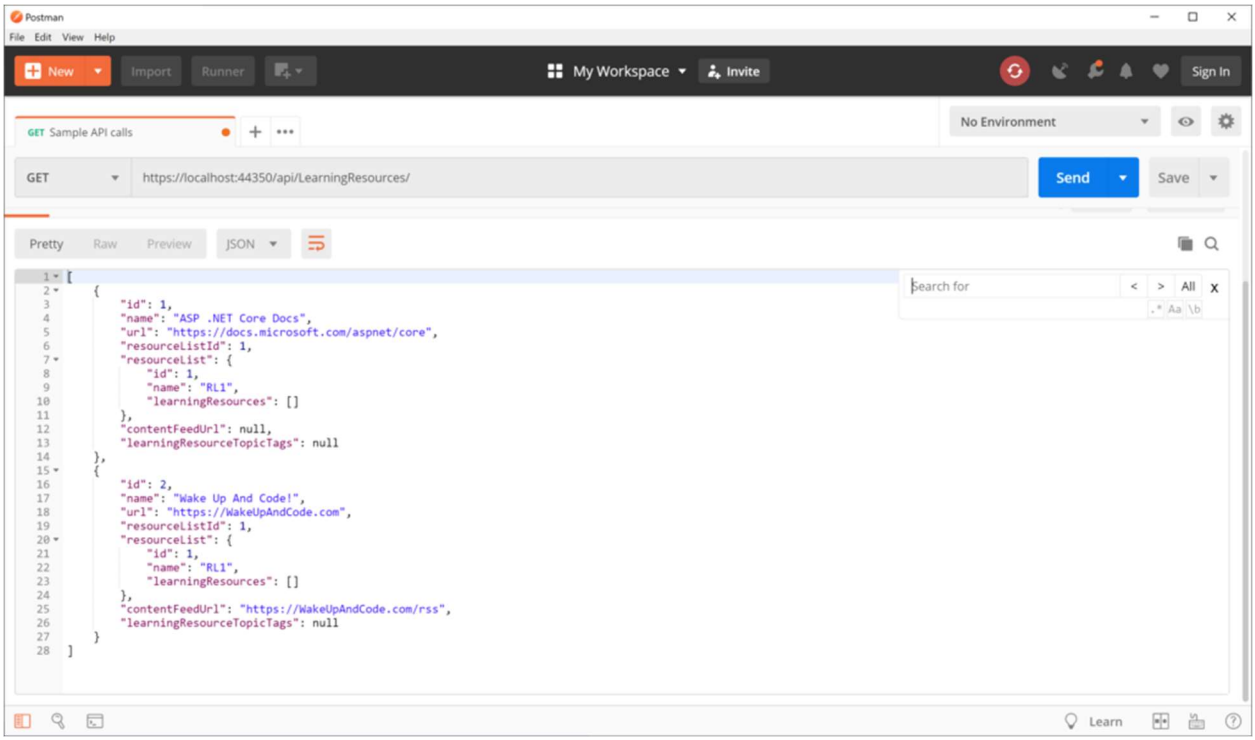
Returning XML/JSON format from a Web API

Returning JsonResult and IActionResult

Before we get into XML output for your Web API, let's start off with JSON output first, and then we'll get to XML. If you run the Web API sample project in the NetLearner repository, you'll notice a LearningResourcesController.cs file that represents a "Learning Resources Controller" that exposes API endpoints. These endpoints can serve up both JSON and XML results of Learning Resources, i.e. blog posts, tutorials, documentation, etc.

Run the application and navigate to the following endpoint in an API testing tool, e.g. Postman:

- https://localhost:44350/api/LearningResources



Sample JSON data in Postman

This triggers a GET request by calling the LearningResourcesController's **Get**() method:

```
 // GET: api/LearningResources
[HttpGet]
public JsonResult Get()
{
    return new JsonResult(_sampleRepository.LearningResources());
}
```

In this case, the **Json**() method returns a **JsonResult** object that serializes a list of Learning Resources. For simplicity, the _sampleRepository object's LearningResources() method (in SampleRepository.cs) returns a hard-coded list of **LearningResource** 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.

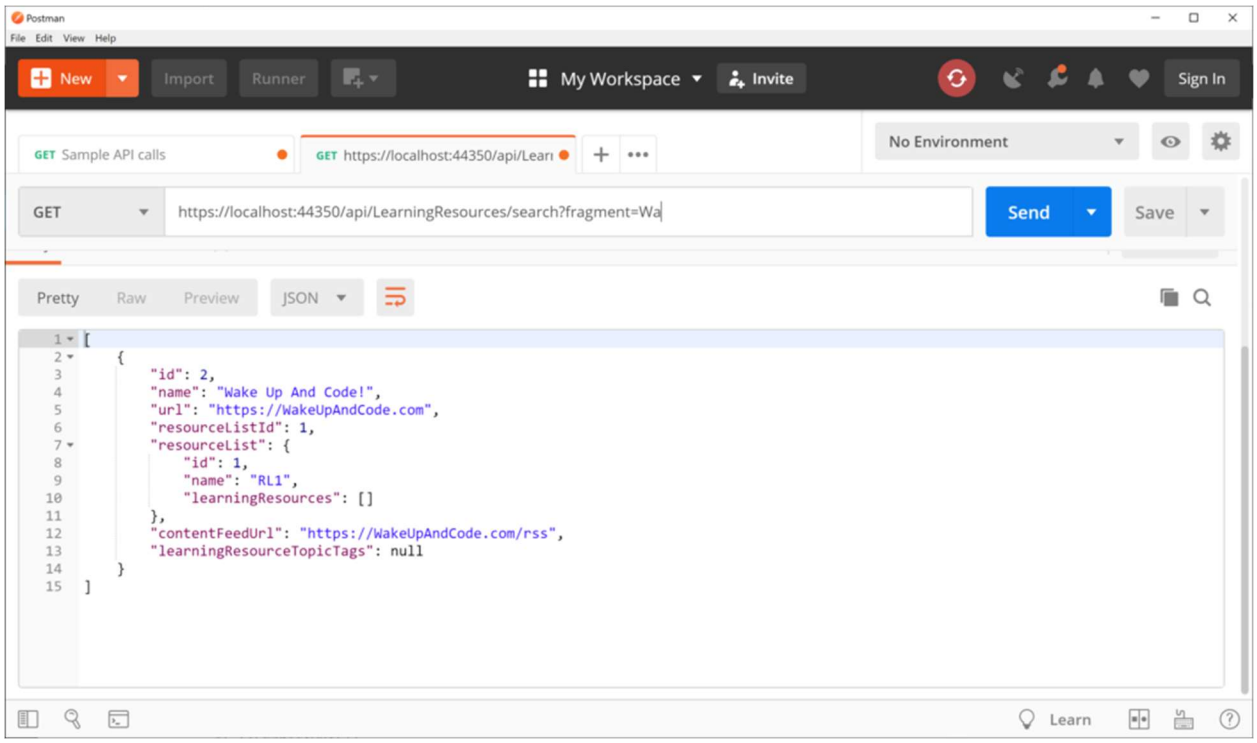
```
public List<LearningResource> LearningResources()
{
   ...
   return new List<LearningResource>
   {
      new LearningResource
      {
         Id= 1,
         Name= "ASP .NET Core Docs",
         Url = "https://docs.microsoft.com/aspnet/core",
         ...
      },
      ...
   }
}
```

The JSON result looks like the following, where a list of learning resources are returned:

```
[
    {
        "id": 1,
        "name": "ASP .NET Core Docs",
        "url": "https://docs.microsoft.com/aspnet/core",
        "resourceListId": 1,
        "resourceList": {
            "id": 1,
            "name": "RL1",
            "learningResources": []
        },
        "contentFeedUrl": null,
        "learningResourceTopicTags": null
    },
    {
        "id": 2,
        "name": "Wake Up And Code!",
        "url": "https://WakeUpAndCode.com",
        "resourceListId": 1,
        "resourceList": {
            "id": 1,
            "name": "RL1",
            "learningResources": []
        },
        "contentFeedUrl": "https://WakeUpAndCode.com/rss",
        "learningResourceTopicTags": null
    }
]
```

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” folllowed by a QueryString parameter “fragment” for a partial match.

- https://localhost:44350/api/LearningResources/**search**?**fragment**=Wa



Sample JSON data with search string

This triggers a GET request by calling the LearningResourceController‘s **Search**() method, with its fragment parameter set to “Wa” for a partial text search:

```
// GET: api/LearningResources/search?fragment=Wa
[HttpGet("Search")]
public IActionResult Search(string fragment)
{
    var result = _sampleRepository.GetByPartialName(fragment);
    if (!result.Any())
    {
        return NotFound(fragment);
    }
    return Ok(result);
}
```

In this case, the **GetByPartialName**() method returns a **List** of **LearningResources** objects that are returned as JSON by default, with an HTTP 200 OK status. In case no results are found, the action method will return a 404 with the **NotFound**() method.

```
public List<LearningResource> GetByPartialName(string nameSubstring)
{
   return LearningResources()
      .Where(lr => lr.Title
         .IndexOf(nameSubstring, 0,
StringComparison.CurrentCultureIgnoreCase) != -1)
      .ToList();
}
```

The JSON result looks like the following, which includes any learning resource that partially matches the string fragment provided:

```
[
    {
        "id": 2,
        "name": "Wake Up And Code!",
        "url": "https://WakeUpAndCode.com",
        "resourceListId": 1,
        "resourceList": {
            "id": 1,
            "name": "RL1",
            "learningResources": []
        },
        "contentFeedUrl": "https://WakeUpAndCode.com/rss",
        "learningResourceTopicTags": null
    }
]
```

Returning Complex Objects

An overloaded version of the **Get**() method takes in a "**listName**" string parameter to filter results by a list name for each learning resource in the repository. Instead of returning a **JsonResult** or **IActionResult**, this one returns a complex object (**LearningResource**) that contains properties that we're interested in.

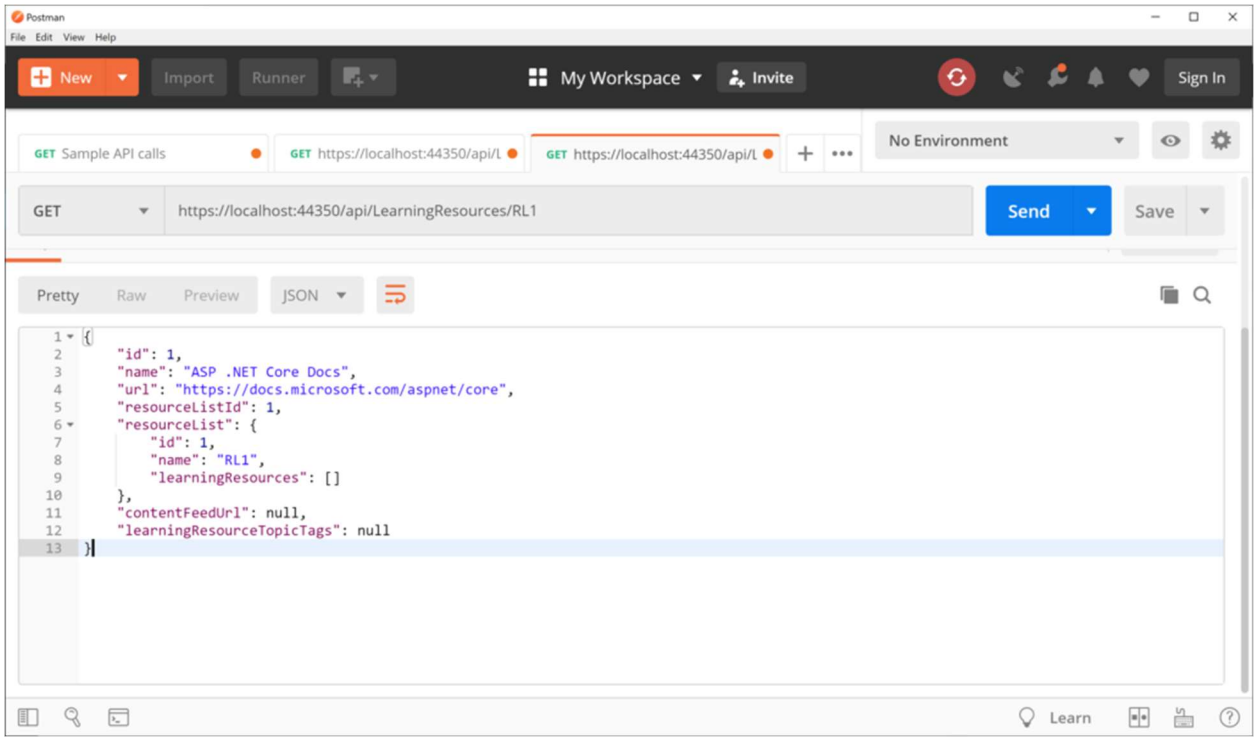
```
// GET api/LearningResources/RL1
[HttpGet("{listName}")]
public LearningResource Get(string listName)
{
   return _sampleRepository.GetByListName(listName);
}
```

The **GetByListName**() method in the SampleRepository.cs class simply checks for a learning resource by the **listName** 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 LearningResource GetByListName(string listName)
{
   return LearningResources().FirstOrDefault(lr =>
lr.ResourceList.Name == listName);
}
```

While the application is running, navigate to the following endpoint:

- https://localhost:44350/api/LearningResources/RL1



Sample JSON data with property filter

This triggers another GET request by calling the LearningResourcesController's overloaded **Get**() method, with the listName parameter. When passing the list name "RL1", this returns one item, as shown below:

```
{
    "id": 1,
    "name": "ASP .NET Core Docs",
    "url": "https://docs.microsoft.com/aspnet/core",
    "resourceListId": 1,
    "resourceList": {
```

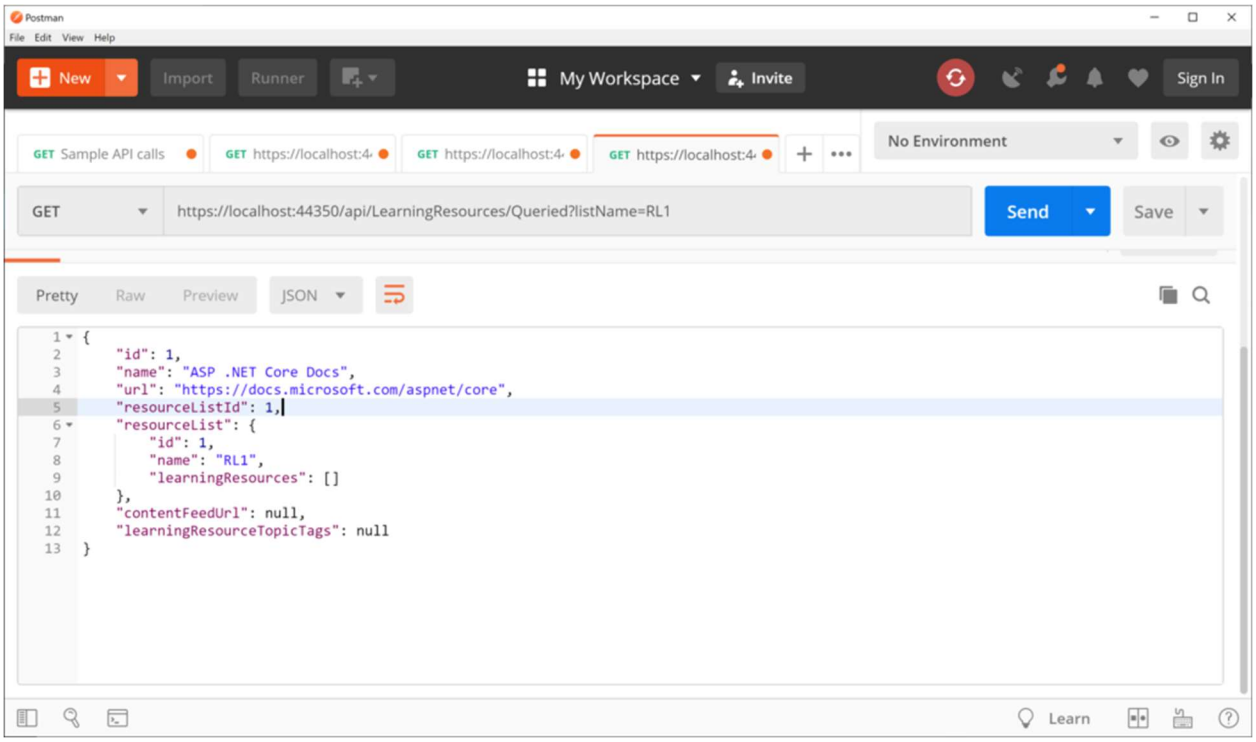
```
        "id": 1,
        "name": "RL1",
        "learningResources": []
    },
    "contentFeedUrl": null,
    "learningResourceTopicTags": null
}
```

Another example with a complex result takes in a similar parameter via QueryString and checks for an exact match with a specific property. In this case the **Queried**() action method calls the repository's existing **GetByListName**() method to find a specific learning resource by its matching list name.

```
// GET: api/LearningResources/queried?listName=RL1
[HttpGet("Queried")]
public LearningResource Queried(string listName)
{
 return _sampleRepository.GetByListName(listName);
}
```

While the application is running, navigate to the following endpoint:

- https://localhost:44350/api/LearningResources/**Queried**?**listName**=RL1



Sample JSON data with QueryString parameter

This triggers a GET request by calling the LearningResourcesController's **Queried**() method, with the **listName** parameter. When passing the list name "RL1", this returns one item, as shown below:

```
{
    "id": 1,
    "name": "ASP .NET Core Docs",
    "url": "https://docs.microsoft.com/aspnet/core",
    "resourceListId": 1,
    "resourceList": {
        "id": 1,
        "name": "RL1",
        "learningResources": []
    },
    "contentFeedUrl": null,
    "learningResourceTopicTags": null
}
```

As you can see, the above result is in JSON format for the returned object.

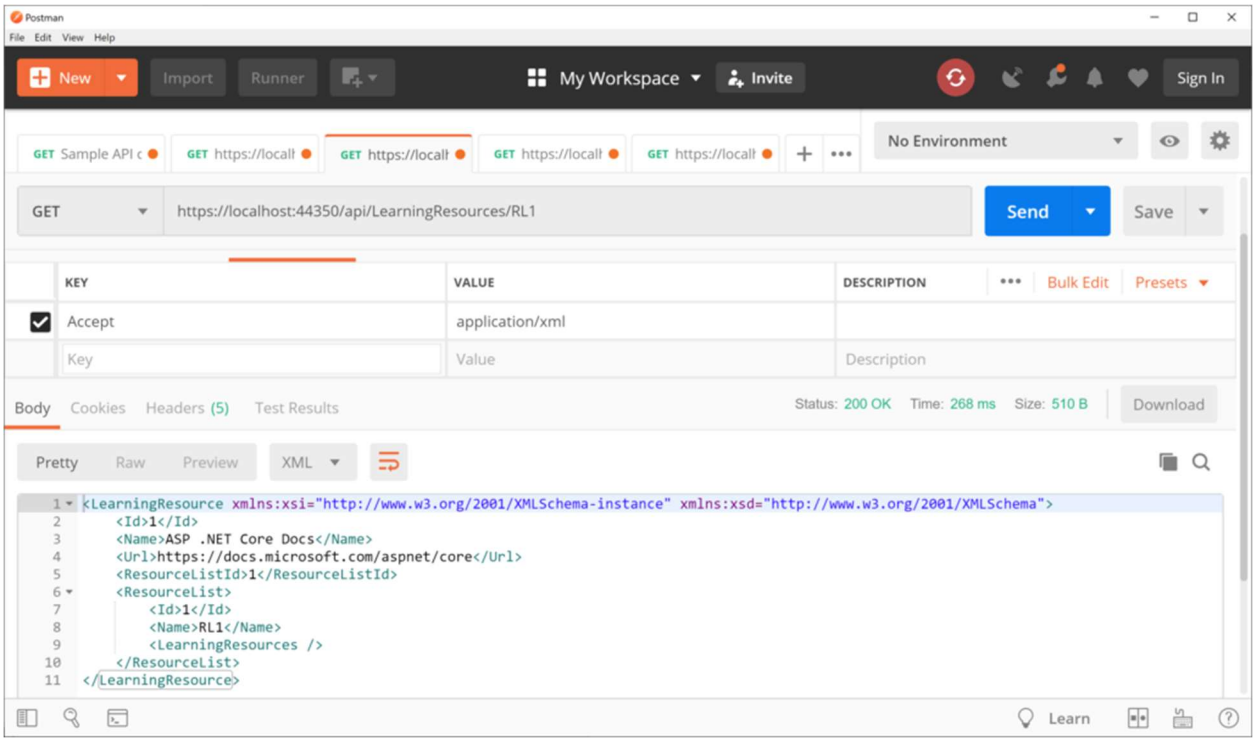
XML Output

Wait a minute… with all these JSON results, when will we get to XML output? Not to worry, there are multiple ways to get XML results while reusing the above code. First, update your Startup.cs file's **ConfigureServices**() to include a call to services.AddControllers().**AddXmlSeralizerFormatters**():

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

In Postman, 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:44350/api/LearningResources/RL1



XML-formatted results in Postman without code changes

This should provide the following XML results:

```
<LearningResource xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <Id>1</Id>
    <Name>ASP .NET Core Docs</Name>
    <Url>https://docs.microsoft.com/aspnet/core</Url>
    <ResourceListId>1</ResourceListId>
    <ResourceList>
        <Id>1</Id>
        <Name>RL1</Name>
        <LearningResources />
    </ResourceList>
</LearningResource>
```

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 API Controller level.

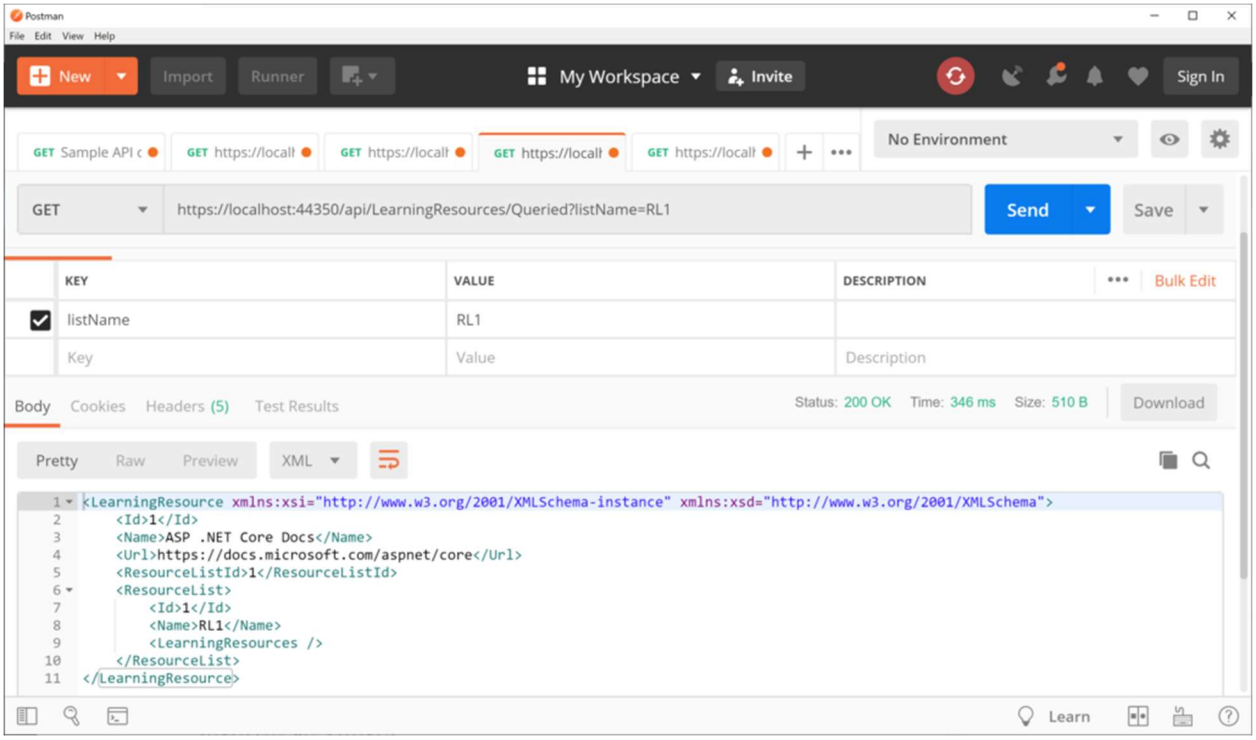
```
[Produces("application/xml")]
[Route("api/[controller]")]
[ApiController]
public class LearningResourcesController : ControllerBase
{
```

```
    . . .
}
```

Then revisit the following endpoint, calling the search action method with the fragment parameter set to “ir”:

- https://localhost:44350/api/LearningResources/**Queried**?**listName**=RL1

At this point, it is no longer necessary to set the Accept header to “application/xml” (in Postman) during the request, since the [**Produces**] attribute is given priority over it.



XML-formatted output using Produces attribute

This should produces the following result , with a **LearningResource** object in XML:

```
<LearningResource xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <Id>1</Id>
    <Name>ASP .NET Core Docs</Name>
    <Url>https://docs.microsoft.com/aspnet/core</Url>
    <ResourceListId>1</ResourceListId>
    <ResourceList>
        <Id>1</Id>
        <Name>RL1</Name>
        <LearningResources />
```

```
    </ResourceList>
</LearningResource>
```

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

- Format response data in ASP.NET Core Web API: https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting
- Postman Reference: https://learning.postman.com/docs/postman/sending-api-requests/requests/

YAML-defined CI/CD for ASP .NET Core 3.1

By Shahed C on June 24, 2020

2 Replies

This is the twenty-fifth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we'll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series will mostly focus on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- XML + JSON Output for Web APIs in ASP .NET Core 3.1

NetLearner on GitHub:

- Repository: https://github.com/shahedc/NetLearnerApp
- v0.25-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.25-alpha

In this Article:

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:

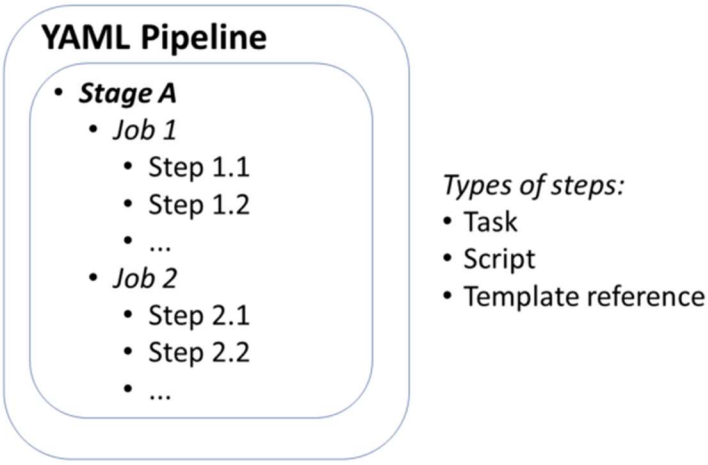
- YAML schema – Azure Pipelines: https://docs.microsoft.com/en-us/azure/devops/pipelines/yaml-schema

In the NetLearner repository, check out the sample YAML code:

- YAML sample: https://github.com/shahedc/NetLearnerApp/blob/main/azure-pipelines.yml.txt

NOTE: Before using the aforementioned YAML sample in an Azure DevOps project, please replace any placeholder values and rename the file to remove the .txt suffix.

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.



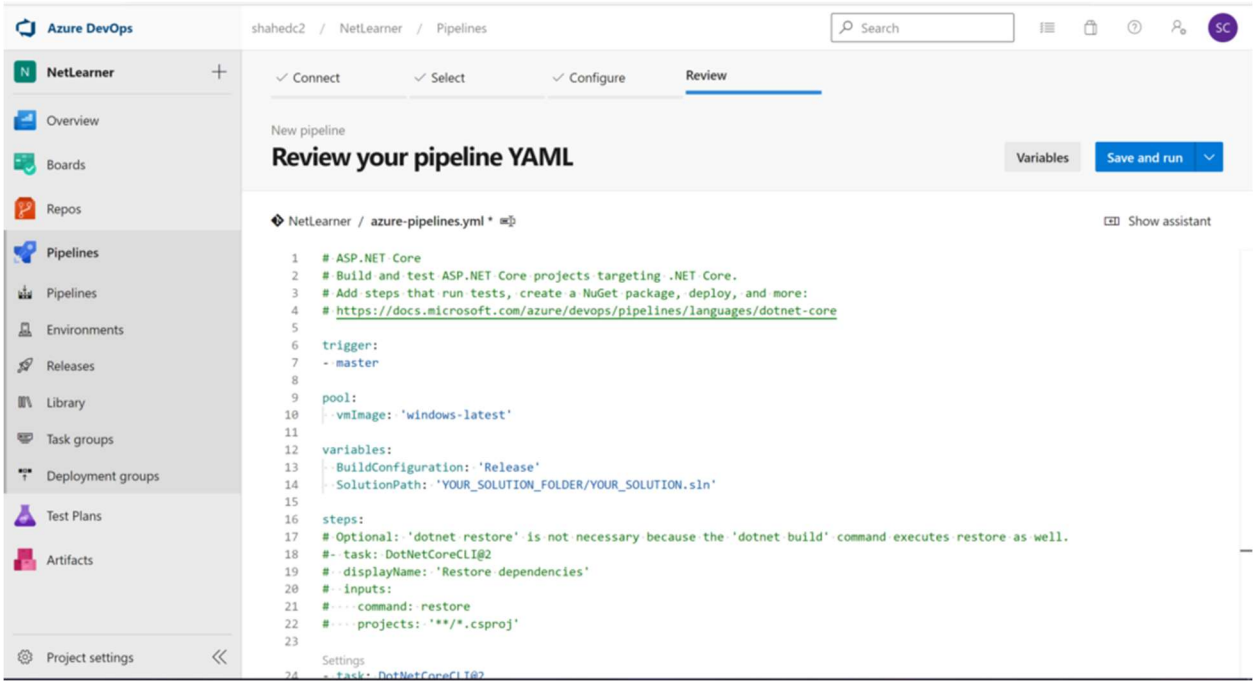
DevOps Pipeline in YAML

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).

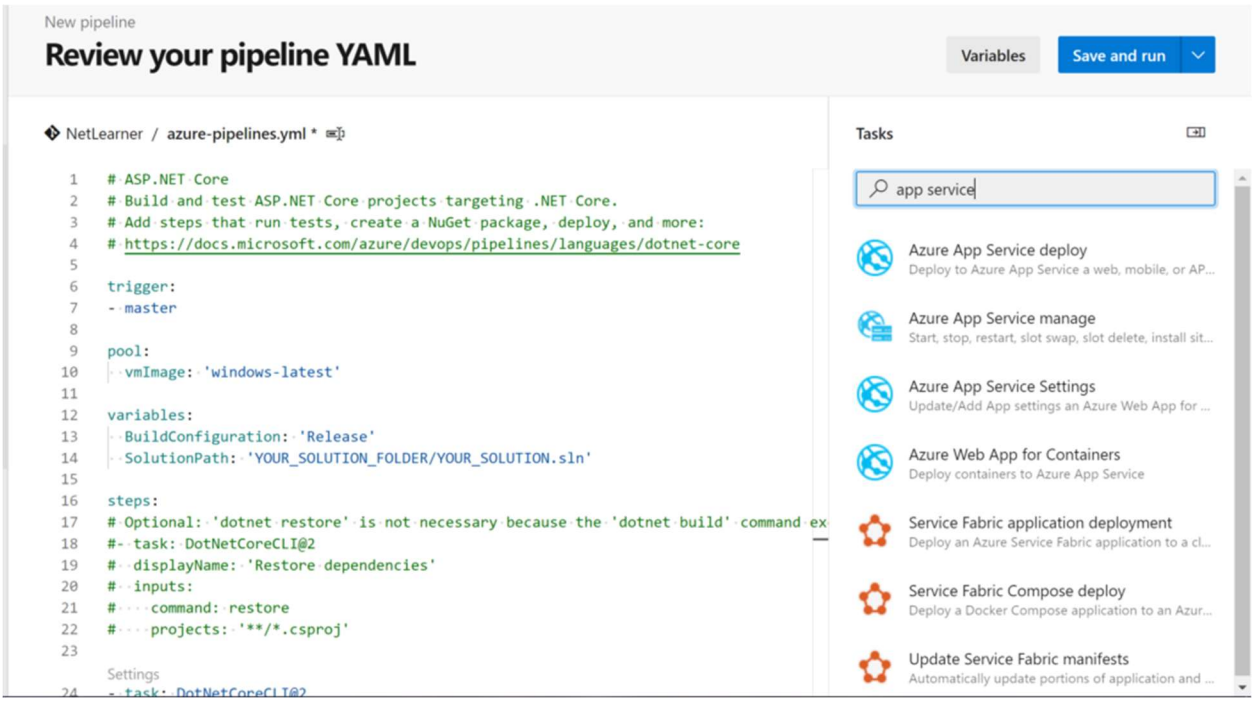


YAML file in Azure DevOps

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/ORy3OeqLZlE

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.



Task Assistant in Azure DevOps

OS/Environment and Runtime

From the sample repo, take a look at the sample YAML code sample “azure-pipelines.yml.txt“. Near the top, there is a definition for a “**pool**” with a “**vmImage**” set to ‘windows-latest’.

```
pool:
 vmImage: 'windows-latest'
```

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.

- Windows Server 2019 with Visual Studio 2019 (windows-latest OR windows-2019)
- Windows Server 2016 with Visual Studio 2017 (vs2017-win2016)
- Ubuntu 18.04 (ubuntu-latest OR ubuntu-18.04)
- Ubuntu 16.04 (ubuntu-16.04)

- macOS X Mojave 10.14 (macOS-10.14)
- macOS X Catalina 10.15 (macOS-latest OR macOS-10.15)

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: '3.1.0'
```

Restore and Build

Once you've set up your OS/environment and runtime, you can restore (dependencies) and build your project. Restoring dependencies with a command is optional since the **Build** step will take care of the **Restore** as well. 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'
  SolutionPath: 'YOUR_SOLUTION_FOLDER/YOUR_SOLUTION.sln'

steps:
# Optional: 'dotnet restore' is not necessary because the 'dotnet
build' command executes restore as well.
#- task: DotNetCoreCLI@2
#  displayName: 'Restore dependencies'
#  inputs:
#    command: restore
#    projects: '**/*.csproj'

- task: DotNetCoreCLI@2
  displayName: 'Build web project'
  inputs:
    command: 'build'
    projects: $(SolutionPath)
```

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: You may also use **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.

```
variables:
 buildConfiguration: 'Release'

steps:
- script: dotnet restore

- script: dotnet build --configuration $(buildConfiguration)
 displayName: 'dotnet build $(buildConfiguration)'
```

From the official docs, here are some more detailed steps for restore and build, if you wish to customize your steps and tasks further:

- Restore: https://docs.microsoft.com/en-us/azure/devops/pipelines/languages/dotnet-core?view=azure-devops#restore-dependencies
- Build: https://docs.microsoft.com/en-us/azure/devops/pipelines/languages/dotnet-core?view=azure-devops#build-your-project

```
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

Publish 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
  displayName: 'Publish and zip'
  inputs:
    command: publish
```

```
    publishWebProjects: False
    projects: $(SolutionPath)
    arguments: '--configuration $(BuildConfiguration) --output
$(Build.ArtifactStagingDirectory)'
    zipAfterPublish: True

- task: AzureWebApp@1
  displayName: 'Deploy Azure Web App'
  inputs:
    azureSubscription: '<REPLACE_WITH_AZURE_SUBSCRIPTION_INFO>'
    appName: <REPLACE_WITH_EXISTING_APP_SERVICE_NAME>
    appType: 'webApp'
    package: $(Build.ArtifactStagingDirectory)/**/*.zip
```

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:

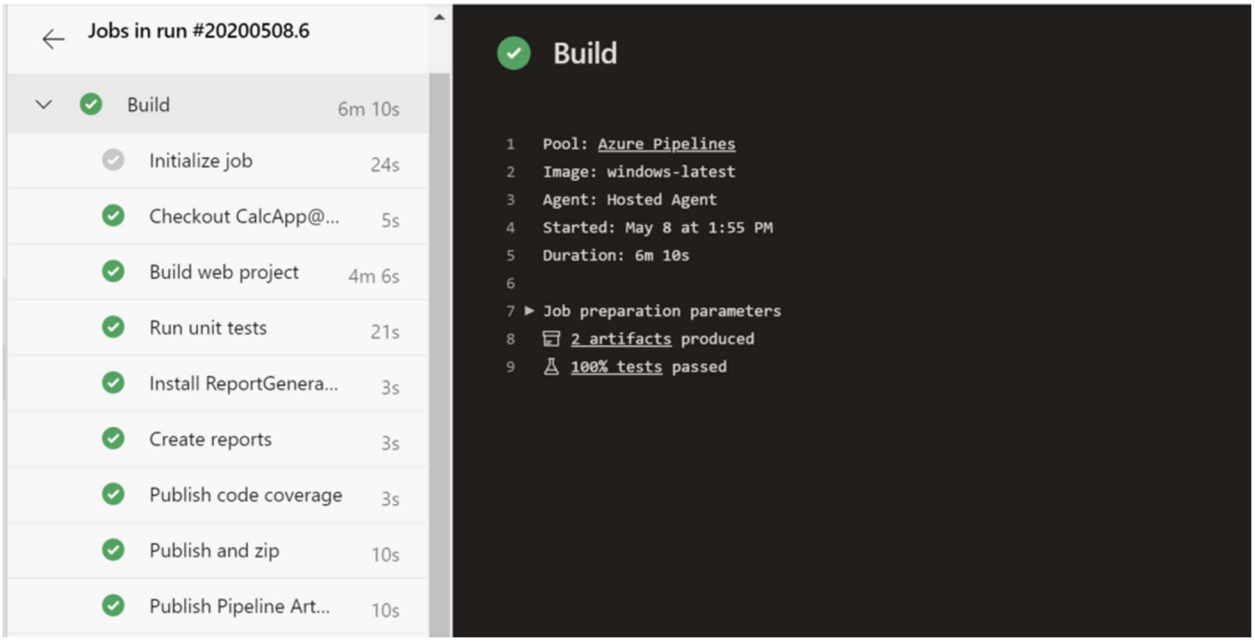
- Predefined variables – Azure Pipelines: https://docs.microsoft.com/en-us/azure/devops/pipelines/build/variables

Note that there is a placeholder text string 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.

The **PublishBuildArtifacts** task uploads the package to a file container, ready for deployment. After your artifacts are ready, a zip file will become available in a named container, e.g. 'drop'.

```
 # Optional step if you want to deploy to some other system using a
Release pipeline or inspect the package afterwards
- task: PublishBuildArtifacts@1
  displayName: 'Publish Build artifacts'
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)'
    ArtifactName: 'drop'
    publishLocation: 'Container'
```

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.



Azure DevOps success messages

NOTE: to set up a release pipeline with multiple stages and optional approval conditions, check out the official docs at:

- 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 correction 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 of the steps and tasks aren't indented correctly.
- **Proper Runtime/OS**: Assign the proper values for the desired runtime, environment and operating system.
- **Publish**: Don't forget to publish 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 may 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=ORy3OeqLZlE
- 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
- YAML for Azure DevOps: https://github.com/shahedc/YamlForAzureDevOps
- Multi-stage YAML Example: https://github.com/shahedc/YamlForAzureDevOps/blob/master/azure-pipelines-multistage.yml

Zero-Downtime* Web Apps for ASP .NET Core 3.1

By Shahed C on June 29, 2020

2 Replies

This is the twenty-sixth of a new series of posts on ASP .NET Core 3.1 for 2020. In this series, we've covered 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the 2019 series, the 2020 series mostly focused on a growing single codebase (NetLearner!) instead of new unrelated code snippets week.

Previous post:

- YAML-defined CI/CD for ASP .NET Core 3.1

NetLearner on GitHub:

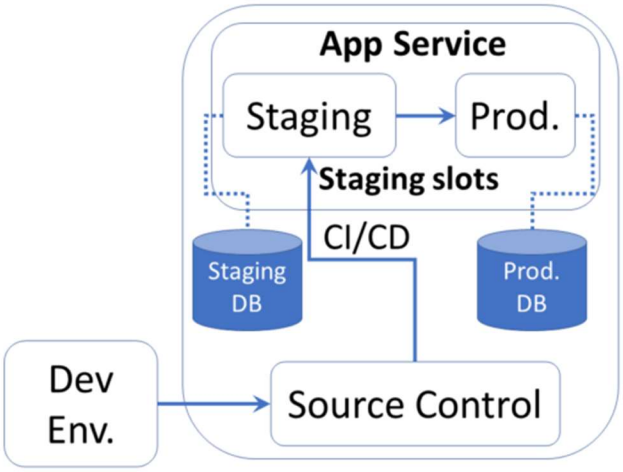
- Repository: https://github.com/shahedc/NetLearnerApp
- v0.26-alpha release: https://github.com/shahedc/NetLearnerApp/releases/tag/v0.26-alpha

In this Article:

- 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.



Deploying to Staging Slots in AsuApp Service

NOTE: 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 when running a web app in multiple Azure regions. In these scenarios, 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

For more information on achieving High Availability (HA) in multiple regions, check out the official docs at:

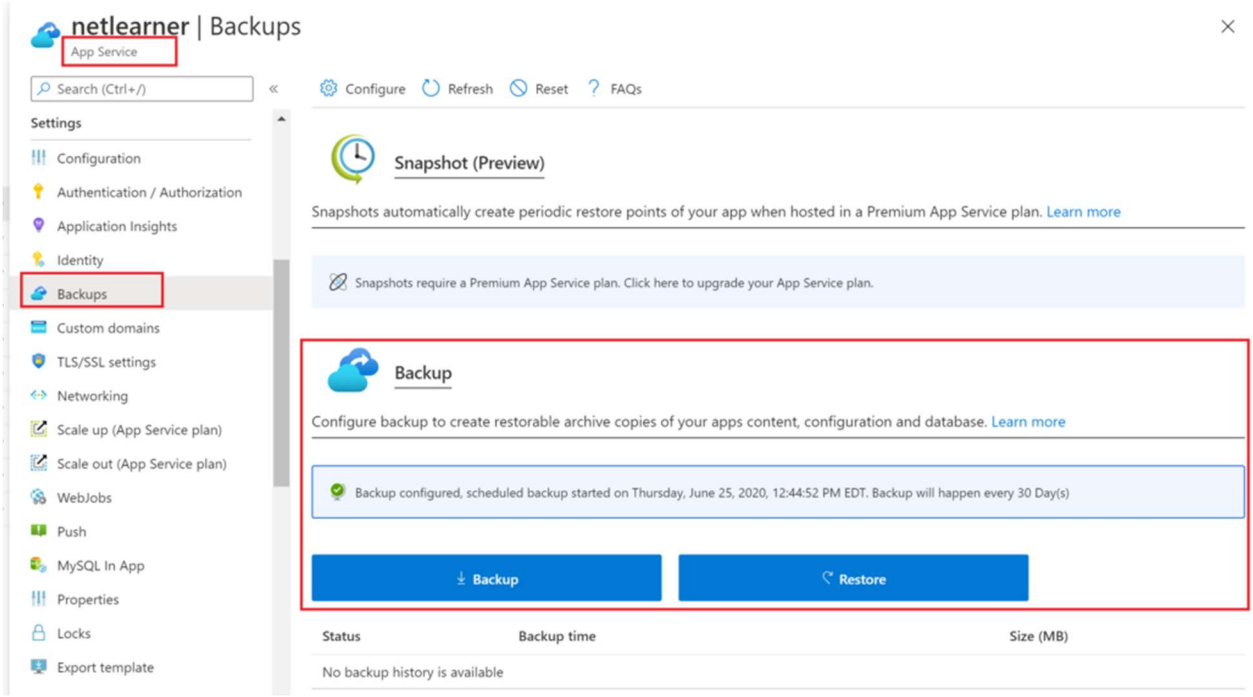
- HA multi-region web apps: https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/app-service-web-app/multi-region

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.



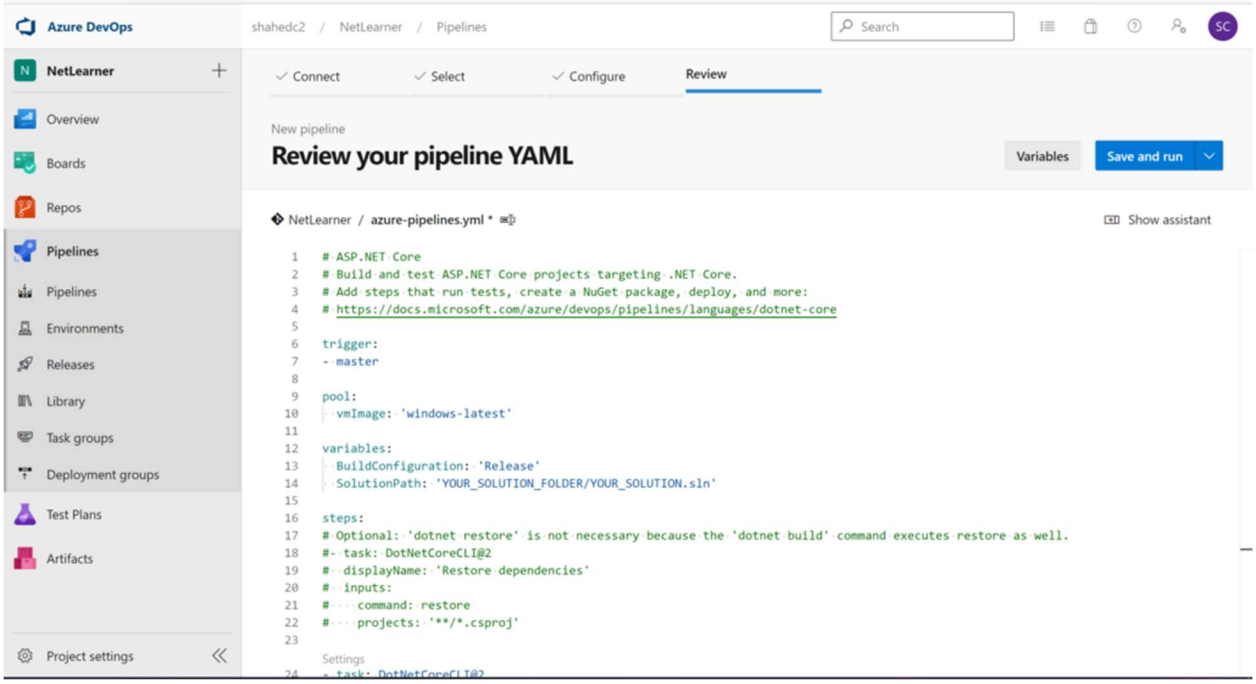
App Service: Backups

NOTE: You may use the new Snapshots feature to *"automatically create perioidic restore points of your app"*. However, this feature is only available in a Premium App Service Plan.

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.

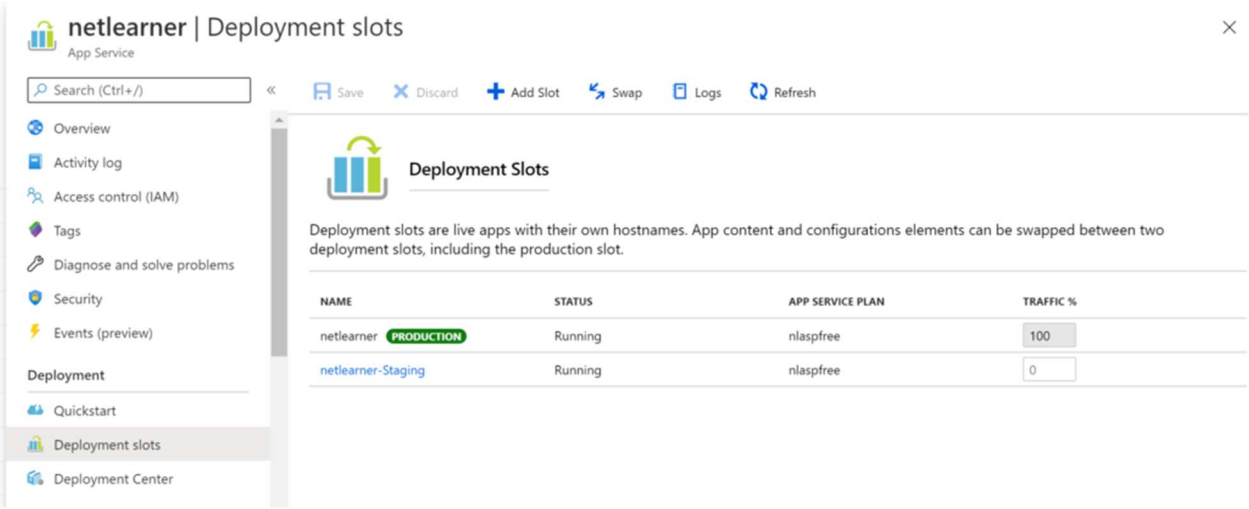


YAML Sample in Azure DevOps

- YAML Sample: https://github.com/shahedc/NetLearnerApp/blob/main/azure-pipelines.yml.txt

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.



Deployment Slots in Azure App Service

Note that Deployment Slots are only available in Standard, Premium or Isolated App Service tiers, not 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 2.0.0
```

If you prefer the Package Manager Console in Visual Studio, you may also use the following PowerShell command:

```
>Install-Package Microsoft.FeatureManagement -Version 2.0.0
```

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