Adopt the IIS in-process hosting model

To adopt the in-process hosting model for IIS, add the `<AspNetCoreHostingModel>` property with a value of `InProcess` to a `<PropertyGroup>` in the project file:

```xml
<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
```

The in-process hosting model isn't supported for ASP.NET Core apps targeting .NET Framework.

For more information, see ASP.NET Core Module (ANCM) for IIS.

Update a custom web.config file

For projects that use a custom *web.config* file in the project root to generate their published *web.config* file:

- In the `<handlers>` entry that adds the ASP.NET Core Module (`name="aspNetCore"`), change the `modules` attribute value from `AspNetCoreModule` to `AspNetCoreModuleV2`.
- In the `<aspNetCore>` element, add the hosting model attribute (`hostingModel="InProcess"`).

For more information and example *web.config* files, see ASP.NET Core Module (ANCM) for IIS.

Update package references

If targeting .NET Core, remove the metapackage reference's `Version` attribute in the project file. Inclusion of a `Version` attribute results in the following warning:

```console
A PackageReference to 'Microsoft.AspNetCore.App' specified a Version of
`2.2.0`. Specifying the version of this package is not recommended. For more
information, see https://aka.ms/sdkimplicitrefs
```

For more information, see Microsoft.AspNetCore.App metapackage for ASP.NET Core.

The metapackage reference should resemble the following `<PackageReference />` node:

XML

```xml
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

If targeting .NET Framework, update each package reference's `Version` attribute to 2.2.0 or later. Here are the package references in a typical ASP.NET Core 2.2 project targeting .NET Framework:

XML

```xml
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.CookiePolicy"
Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.HttpsPolicy"
Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles"
Version="2.2.0" />
</ItemGroup>
```

If referencing the Microsoft.AspNetCore.Razor.Design package, update its `Version` attribute to 2.2.0 or later. Failure to do so results in the following error:

Console

```
Detected package downgrade: Microsoft.AspNetCore.Razor.Design from 2.2.0 to
2.1.2. Reference the package directly from the project to select a different
version.
```

Update .NET Core SDK version in global.json

If your solution relies upon a global.json file to target a specific .NET Core SDK version, update its `version` property to the 2.2 version installed on your machine:

JSON

```json
{
  "sdk": {
    "version": "2.2.100"
  }
}
```

Update launch settings

If using Visual Studio Code, update the project's launch settings file (`.vscode/launch.json`). The `program` path should reference the new TFM:

JSON

```json
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "program": "${workspaceFolder}/bin/Debug/netcoreapp2.2/test-
app.dll",
            "args": [],
            "cwd": "${workspaceFolder}",
            "stopAtEntry": false,
            "internalConsoleOptions": "openOnSessionStart",
            "launchBrowser": {
                "enabled": true,
                "args": "${auto-detect-url}",
                "windows": {
                    "command": "cmd.exe",
                    "args": "/C start ${auto-detect-url}"
                },
                "osx": {
                    "command": "open"
                },
                "linux": {
                    "command": "xdg-open"
                }
            },
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            },
            "sourceFileMap": {
                "/Views": "${workspaceFolder}/Views"
            }
        },
        {
            "name": ".NET Core Attach",
            "type": "coreclr",
            "request": "attach",
            "processId": "${command:pickProcess}"
        }
    ]
}
```

Update Kestrel configuration

If the app calls UseKestrel by calling `CreateDefaultBuilder` in the CreateWebHostBuilder method of the `Program` class, call `ConfigureKestrel` to configure Kestrel server instead of `UseKestrel` in order to avoid conflicts with the IIS in-process hosting model:

C#

```csharp
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            // Set properties and call methods on options
        });
```

If the app doesn't call `CreateDefaultBuilder` and builds the host manually in the `Program` class, call UseKestrel **before** calling `ConfigureKestrel`:

C#

```csharp
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseKestrel()
        .UseIISIntegration()
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            // Set properties and call methods on options
        })
        .Build();

    host.Run();
}
```

For more information, see Kestrel web server in ASP.NET Core.

Update compatibility version

Update the compatibility version in `Startup.ConfigureServices` to `Version_2_2`:

C#

```csharp
services.AddMvc()
```

```
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

Update CORS policy

In ASP.NET Core 2.2, the CORS middleware responds with a wildcard origin (`*`) if a policy allows any origin and allows credentials. Credentials aren't supported when a wildcard origin (`*`) is specified, and browsers will disallow the CORS request. For more information, including options for correcting the problem on the client, see the MDN web docs.

To correct this problem on the server, take one of the following actions:

- Modify the CORS policy to no longer allow credentials. That is, remove the call to AllowCredentials when configuring the policy.
- If credentials are required for the CORS request to succeed, modify the policy to specify allowed hosts. For example, use `builder.WithOrigins("https://api.example1.com", "https://example2.com")` instead of using AllowAnyOrigin.

Update Docker images

The following table shows the Docker image tag changes:

Expand table

2.1	2.2
`microsoft/dotnet:2.1-aspnetcore-runtime`	`mcr.microsoft.com/dotnet/core/aspnet:2.2`
`microsoft/dotnet:2.1-sdk`	`mcr.microsoft.com/dotnet/core/sdk:2.2`

Change the `FROM` lines in your *Dockerfile* to use the new image tags in the preceding table's 2.2 column.

Build manually in Visual Studio when using IIS in-process hosting

Visual Studio's **Auto build on browser request** experience doesn't function with the IIS in-process hosting model. You must manually rebuild the project when using in-process hosting. Improvements to this experience are planned for a future release of Visual Studio.

Update logging code

Recommended logging configuration code didn't change from 2.1 to 2.2, but some 1.x coding patterns that still worked in 2.1 no longer work in 2.2.

If your app does logging provider initialization, filtering, and configuration loading in the `Startup` class, move that code to `Program.Main`:

- Provider initialization:

  1.x example:

  C#

  ```csharp
  public void Configure(IApplicationBuilder app, ILoggerFactory
  loggerFactory)
  {
      loggerFactory.AddConsole();
  }
  ```

  2.2 example:

  C#

  ```csharp
  public static void Main(string[] args)
  {
      var webHost = new WebHostBuilder()
          // ...
          .ConfigureLogging((hostingContext, logging) =>
          {
              logging.AddConsole();
          })
          // ...
  }
  ```

- Filtering:

  1.x example:

  C#

  ```csharp
  public void Configure(IApplicationBuilder app, ILoggerFactory
  loggerFactory)
  ```

```
{
    loggerFactory.AddConsole(LogLevel.Information);
    // or
    loggerFactory.AddConsole((category, level) =>
        category == "A" || level == LogLevel.Critical);
}
```

2.2 example:

```C#
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        // ...
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.AddConsole()
                   .AddFilter<ConsoleLoggerProvider>
                       (category: null, level: LogLevel.Information)
                   // or
                   .AddFilter<ConsoleLoggerProvider>
                       ((category, level) => category == "A" ||
                           level == LogLevel.Critical)
            );
        })
        // ...
}
```

- Configuration loading:

  1.x example:

```C#
public void Configure(IApplicationBuilder app, ILoggerFactory
loggerFactory)
{
    loggerFactory.AddConsole(Configuration);
}
```

  2.2 example:

```C#
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        // ...
        .ConfigureLogging((hostingContext, logging) =>
        {

logging.AddConfiguration(hostingContext.Configuration.GetSection("Loggi
ng"));
            logging.AddConsole();
        })
        // ...
}
```

For more information, see Logging in .NET Core and ASP.NET Core

ASP.NET Core Module (ANCM)

If the ASP.NET Core Module (ANCM) wasn't a selected component when Visual Studio was installed or if a prior version of the ANCM was installed on the system, download the latest .NET Core Hosting Bundle Installer (direct download) and run the installer. For more information, see Hosting Bundle.

Additional resources

- Compatibility version for ASP.NET Core MVC
- Microsoft.AspNetCore.App metapackage for ASP.NET Core
- Implicit package references

Migrate from Microsoft.Extensions.Logging 2.1 to 2.2 or 3.0

Article • 06/03/2022

This article outlines the common steps for migrating a non-ASP.NET Core application that uses `Microsoft.Extensions.Logging` from 2.1 to 2.2 or 3.0.

2.1 to 2.2

Manually create `ServiceCollection` and call `AddLogging`.

2.1 example:

```C#
using (var loggerFactory = new LoggerFactory())
{
    loggerFactory.AddConsole();

    // use loggerFactory
}
```

2.2 example:

```C#
var serviceCollection = new ServiceCollection();
serviceCollection.AddLogging(builder => builder.AddConsole());

using (var serviceProvider = serviceCollection.BuildServiceProvider())
using (var loggerFactory = serviceProvider.GetService<ILoggerFactory>())
{
    // use loggerFactory
}
```

2.1 to 3.0

In 3.0, use `LoggingFactory.Create`.

2.1 example:

```C#
using (var loggerFactory = new LoggerFactory())
{
    loggerFactory.AddConsole();

    // use loggerFactory
}
```

3.0 example:

```C#
using (var loggerFactory = LoggerFactory.Create(builder =>
builder.AddConsole()))
{
    // use loggerFactory
}
```

Additional resources

- Microsoft.Extensions.Logging.Console NuGet package.
- Logging in .NET Core and ASP.NET Core

Migrate from ASP.NET Core 2.0 to 2.1

Article • 08/30/2024

By Rick Anderson

See What's new in ASP.NET Core 2.1 for an overview of the new features in ASP.NET Core 2.1.

This article:

- Covers the basics of migrating an ASP.NET Core 2.0 app to 2.1.
- Provides an overview of the changes to the ASP.NET Core web application templates.

A quick way to get an overview of the changes in 2.1 is to:

- Create an ASP.NET Core 2.0 web app named WebApp1.
- Commit the WebApp1 in a source control system.
- Delete WebApp1 and create an ASP.NET Core 2.1 web app named WebApp1 in the same place.
- Review the changes in the 2.1 version.

This article provides an overview on migration to ASP.NET Core 2.1. It doesn't contain a complete list of all changes needed to migrate to version 2.1. Some projects might require more steps depending on the options selected when the project was created and modifications made to the project.

Update the project file to use 2.1 versions

Update the project file:

- Change the target framework to .NET Core 2.1 by updating the project file to `<TargetFramework>netcoreapp2.1</TargetFramework>`.
- Replace the package reference for `Microsoft.AspNetCore.All` with a package reference for `Microsoft.AspNetCore.App`. You may need to add dependencies that were removed from `Microsoft.AspNetCore.All`. For more information, see Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.0 and Microsoft.AspNetCore.App metapackage for ASP.NET Core.
- Remove the "Version" attribute on the package reference to `Microsoft.AspNetCore.App`. Projects that use `<Project Sdk="Microsoft.NET.Sdk.Web">` don't need to set the version. The version is implied

by the target framework and selected to best match the way ASP.NET Core 2.1 works. For more information, see the Rules for projects targeting the shared framework section.

- For apps that target the .NET Framework, update each package reference to 2.1.
- Remove references to **<DotNetCliToolReference>** elements for the following packages. These tools are bundled by default in the .NET CLI and don't need to be installed separately.
  - Microsoft.DotNet.Watcher.Tools (`dotnet watch`)
  - Microsoft.EntityFrameworkCore.Tools.DotNet (`dotnet ef`)
  - Microsoft.Extensions.Caching.SqlConfig.Tools (`dotnet sql-cache`)
  - Microsoft.Extensions.SecretManager.Tools (`dotnet user-secrets`)
- Optional: you can remove the **<DotNetCliToolReference>** element for `Microsoft.VisualStudio.Web.CodeGeneration.Tools`. You can replace this tool with a globally installed version by running `dotnet tool install -g dotnet-aspnet-codegenerator`.
- For 2.1, a Razor class library is the recommended solution to distribute Razor files. If your app uses embedded views, or otherwise relies on runtime compilation of Razor files, add `<CopyRefAssembliesToPublishDirectory>true</CopyRefAssembliesToPublishDirectory>` to a `<PropertyGroup>` in your project file.

The following markup shows the template-generated 2.0 project file:

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <UserSecretsId>aspnet-{Project Name}-{GUID}</UserSecretsId>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="2.0.3" PrivateAssets="All" />
    <PackageReference
Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.4"
PrivateAssets="All" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference
Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.3" />
    <DotNetCliToolReference
Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.2" />
    <DotNetCliToolReference
Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.4" />
```

```
  </ItemGroup>
</Project>
```

The following markup shows the template-generated 2.1 project file:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <UserSecretsId>aspnet-{Project Name}-{GUID}</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference 
Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.1.1" 
PrivateAssets="All" />
  </ItemGroup>

</Project>
```

Rules for projects targeting the shared framework

A *shared framework* is a set of assemblies (*.dll* files) that are not in the app's folders. The shared framework must be installed on the machine to run the app. For more information, see The shared framework.

ASP.NET Core 2.1 includes the following shared frameworks:

- Microsoft.AspNetCore.App
- Microsoft.AspNetCore.All

The version specified by the package reference is the *minimum required* version. For example, a project referencing the 2.1.1 versions of these packages won't run on a machine with only the 2.1.0 runtime installed.

Known issues for projects targeting a shared framework:

- The .NET Core 2.1.300 SDK (first included in Visual Studio 15.6) set the implicit version of `Microsoft.AspNetCore.App` to 2.1.0 which caused conflicts with Entity Framework Core 2.1.1. The recommended solution is to upgrade the .NET Core

SDK to 2.1.301 or later. For more information, see Packages that share dependencies with Microsoft.AspNetCore.App cannot reference patch versions.

- All projects that must use `Microsoft.AspNetCore.All` or `Microsoft.AspNetCore.App` should add a package reference for the package in the project file, even if they contain a project reference to another project using `Microsoft.AspNetCore.All` or `Microsoft.AspNetCore.App`.

  Example:
  - `MyApp` has a package reference to `Microsoft.AspNetCore.App`.
  - `MyApp.Tests` has a project reference to `MyApp.csproj`.

  Add a package reference for `Microsoft.AspNetCore.App` to `MyApp.Tests`. For more information, see Integration testing is hard to set up and may break on shared framework servicing.

Update to the 2.1 Docker images

In ASP.NET Core 2.1, the Docker images migrated to the dotnet/dotnet-docker GitHub repository. The following table shows the Docker image and tag changes:

Expand table

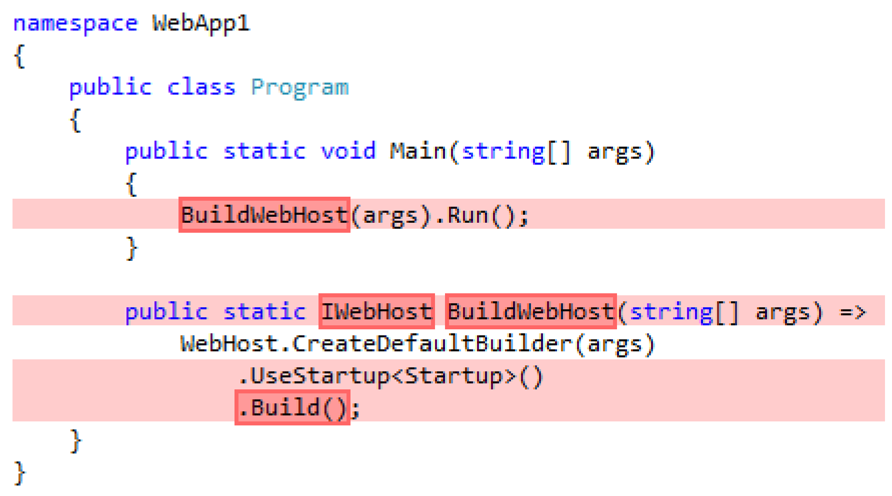
2.0	2.1
microsoft/aspnetcore:2.0	microsoft/dotnet:2.1-aspnetcore-runtime
microsoft/aspnetcore-build:2.0	microsoft/dotnet:2.1-sdk

Change the `FROM` lines in your *Dockerfile* to use the new image names and tags in the preceding table's 2.1 column. For more information, see Migrating from aspnetcore docker repos to dotnet.

Changes to take advantage of the new code-based idioms that are recommended in ASP.NET Core 2.1

Changes to Main

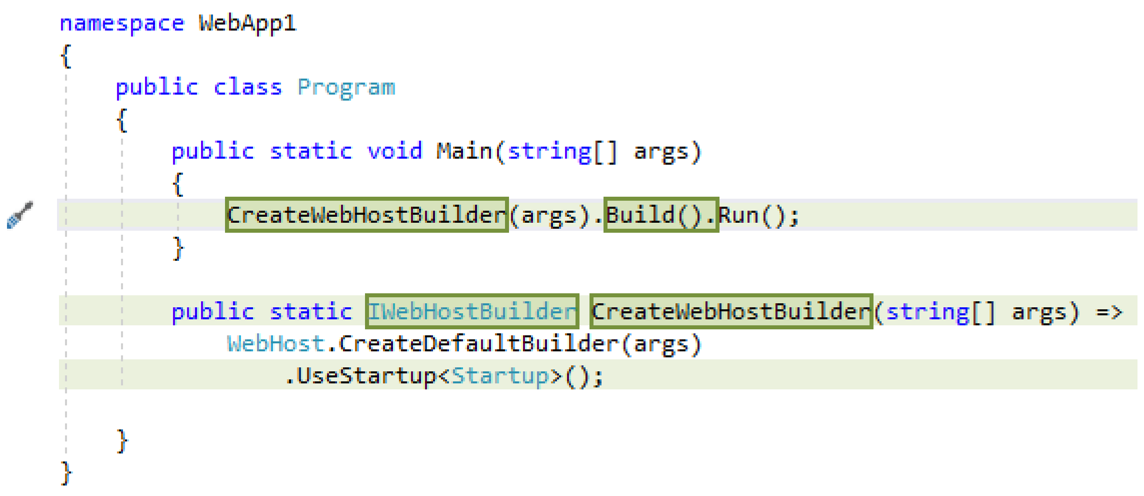
The following images show the changes made to the templated generated `Program.cs` file.



```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

The preceding image shows the 2.0 version with the deletions in red.

The following image shows the 2.1 code. The code in green replaced the 2.0 version:



```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

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

    }
}
```

The following code shows the 2.1 version of `Program.cs`:

C#

```csharp
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

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

The new `Main` replaces the call to `BuildWebHost` with CreateWebHostBuilder. IWebHostBuilder was added to support a new integration test infrastructure.

Changes to Startup

The following code shows the changes to 2.1 template generated code. All changes are newly added code, except that `UseBrowserLink` has been removed:

```C#
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace WebApp1
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        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;
            });


            services.AddMvc()
                .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
        }

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

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            // If the app uses Session or TempData based on Session:
            // app.UseSession();

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

The preceding code changes are detailed in:

- GDPR support in ASP.NET Core for `CookiePolicyOptions` and `UseCookiePolicy`.
- HTTP Strict Transport Security Protocol (HSTS) for `UseHsts`.
- Require HTTPS for `UseHttpsRedirection`.
- SetCompatibilityVersion for `SetCompatibilityVersion(CompatibilityVersion.Version_2_1)`.

Changes to authentication code

ASP.NET Core 2.1 provides ASP.NET Core Identity as a Razor class library (RCL).

The default 2.1 Identity UI doesn't currently provide significant new features over the 2.0 version. Replacing Identity with the RCL package is optional. The advantages to replacing the template generated Identity code with the RCL version include:

- Many files are moved out of your source tree.
- Any bug fixes or new features to Identity are included in the Microsoft.AspNetCore.App metapackage. You automatically get the updated Identity when `Microsoft.AspNetCore.App` is updated.

If you've made non-trivial changes to the template generated Identity code:

- The preceding advantages probably do **not** justify converting to the RCL version.
- You can keep your ASP.NET Core 2.0 Identity code, it's fully supported.

Identity 2.1 exposes endpoints with the `Identity` area. For example, the follow table shows examples of Identity endpoints that change from 2.0 to 2.1:

Expand table

2.0 URL	2.1 URL
/Account/Login	/Identity/Account/Login
/Account/Logout	/Identity/Account/Logout
/Account/Manage	/Identity/Account/Manage

Applications that have code using Identity and replace 2.0 Identity UI with the 2.1 Identity Library need to take into account Identity URLs have `/Identity` segment prepended to the URIs. One way to handle the new Identity endpoints is to set up redirects, for example from `/Account/Login` to `/Identity/Account/Login`.

Update Identity to version 2.1

The following options are available to update Identity to 2.1.

- Use the Identity UI 2.0 code with no changes. Using Identity UI 2.0 code is fully supported. This is a good approach when significant changes have been made to the generated Identity code.
- Delete your existing Identity 2.0 code and Scaffold Identity into your project. Your project will use the ASP.NET Core Identity Razor class library. You can generate code and UI for any of the Identity UI code that you modified. Apply your code changes to the newly scaffolded UI code.
- Delete your existing Identity 2.0 code and Scaffold Identity into your project with the option to **Override all files**.

Replace Identity 2.0 UI with the Identity 2.1 Razor class library

This section outlines the steps to replace the ASP.NET Core 2.0 template-generated Identity code with the ASP.NET Core Identity Razor class library. The following steps are for a Razor Pages project, but the approach for an MVC project is similar.

- Verify the project file is updated to use 2.1 versions
- Delete the following folders and all the files in them:
  - *Controllers*
  - *Pages/Account/*

- *Extensions*
- Build the project.
- Scaffold Identity into your project:
  - Select the projects exiting *_Layout.cshtml* file.
  - Select the + icon on the right side of the **Data context class**. Accept the default name.
  - Select **Add** to create a new Data context class. Creating a new data context is required for to scaffold. You remove the new data context in the next section.

Update after scaffolding Identity

- Delete the Identity scaffolder generated `IdentityDbContext` derived class in the *Areas/Identity/Data/* folder.

- Delete `Areas/Identity/IdentityHostingStartup.cs`.

- Update the *_LoginPartial.cshtml* file:
  - Move *Pages/_LoginPartial.cshtml* to *Pages/Shared/_LoginPartial.cshtml*.
  - Add `asp-area="Identity"` to the form and anchor links.
  - Update the `<form />` element to `<form asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Page("/Index", new { area = "" })" method="post" id="logoutForm" class="navbar-right">`.

  The following code shows the updated *_LoginPartial.cshtml* file:

  CSHTML

```cshtml
@using Microsoft.AspNetCore.Identity

@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="Identity" asp-page="/Account/Logout" asp-route-
returnUrl="@Url.Page("/Index", new { area = "" })" method="post"
id="logoutForm" class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="Identity" asp-page="/Account/Manage/Index"
title="Manage">Hello @UserManager.GetUserName(User)!</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn
navbar-link">Log out</button>
            </li>
        </ul>
    </form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="Identity" asp-
page="/Account/Register">Register</a></li>
        <li><a asp-area="Identity" asp-page="/Account/Login">Log in</a>
</li>
    </ul>
}
```

Update `ConfigureServices` with the following code:

```C#
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))
);

    services.AddDefaultIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Register no-op EmailSender used by account confirmation and password
reset
    // during development
    services.AddSingleton<IEmailSender, EmailSender>();
}
```

Changes to Razor Pages projects Razor files

The layout file

- Move *Pages/_Layout.cshtml* to *Pages/Shared/_Layout.cshtml*
- In *Areas/Identity/Pages/_ViewStart.cshtml*, change `Layout = "/Pages/_Layout.cshtml"` to `Layout = "/Pages/Shared/_Layout.cshtml"`.
- The *_Layout.cshtml* file has the following changes:
    - `<partial name="_CookieConsentPartial" />` is added. For more information, see GDPR support in ASP.NET Core.

    - jQuery changes from 2.2.0 to 3.3.1.

_ValidationScriptsPartial.cshtml

- *Pages/_ValidationScriptsPartial.cshtml* moves to *Pages/Shared/_ValidationScriptsPartial.cshtml*.
- *jquery.validate/1.14.0* changes to *jquery.validate/1.17.0*.

New files

The following files are added:

- `Privacy.cshtml`
- `Privacy.cshtml.cs`

See GDPR support in ASP.NET Core for information on the preceding files.

Changes to MVC projects Razor files

The layout file

The `Layout.cshtml` file has the following changes:

- `<partial name="_CookieConsentPartial" />` is added.
- jQuery changes from 2.2.0 to 3.3.1

_ValidationScriptsPartial.cshtml

jquery.validate/1.14.0 changes to *jquery.validate/1.17.0*

New files and action methods

The following are added:

- `Views/Home/Privacy.cshtml`
- The `Privacy` action method is added to the Home controller.

See GDPR support in ASP.NET Core for information on the preceding files.

Changes to the launchSettings.json file

As ASP.NET Core apps now use HTTPS by default, the `Properties/launchSettings.json` file has changed.

The following JSON shows the earlier 2.0 template-generated `launchSettings.json` file:

JSON

```json
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:1799/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApp1": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:1798/"
    }
  }
}
```

The following JSON shows the new 2.1 template-generated `launchSettings.json` file:

JSON

```json
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:39191",
      "sslPort": 44390
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApp1": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

For more information, see Enforce HTTPS in ASP.NET Core.

Breaking changes

FileResult Range header

FileResult no longer processes the Accept-Ranges header by default. To enable the `Accept-Ranges` header, set EnableRangeProcessing to `true`.

ControllerBase.File and PhysicalFile Range header

The following ControllerBase methods no longer processes the Accept-Ranges header by default:

- Overloads of ControllerBase.File
- ControllerBase.PhysicalFile

To enable the `Accept-Ranges` header, set the `EnableRangeProcessing` parameter to `true`.

ASP.NET Core Module (ANCM)

If the ASP.NET Core Module (ANCM) wasn't a selected component when Visual Studio was installed or if a prior version of the ANCM was installed on the system, download the latest .NET Core Hosting Bundle Installer (direct download) and run the installer. For more information, see Hosting Bundle.

Additional changes

- SetCompatibilityVersion
- Libuv transport configuration

Migrate from ASP.NET Core 1.x to 2.0

Article • 06/18/2024

By Scott Addie

In this article, we walk you through updating an existing ASP.NET Core 1.x project to ASP.NET Core 2.0. Migrating your application to ASP.NET Core 2.0 enables you to take advantage of many new features and performance improvements.

Existing ASP.NET Core 1.x applications are based off of version-specific project templates. As the ASP.NET Core framework evolves, so do the project templates and the starter code contained within them. In addition to updating the ASP.NET Core framework, you need to update the code for your application.

Prerequisites

See Get Started with ASP.NET Core.

Update Target Framework Moniker (TFM)

Projects targeting .NET Core should use the TFM of a version greater than or equal to .NET Core 2.0. Search for the `<TargetFramework>` node in the `.csproj` file, and replace its inner text with `netcoreapp2.0`:

XML

```xml
<TargetFramework>netcoreapp2.0</TargetFramework>
```

Projects targeting .NET Framework should use the TFM of a version greater than or equal to .NET Framework 4.6.1. Search for the `<TargetFramework>` node in the `.csproj` file, and replace its inner text with `net461`:

XML

```xml
<TargetFramework>net461</TargetFramework>
```

Note

.NET Core 2.0 offers a much larger surface area than .NET Core 1.x. If you're targeting .NET Framework solely because of missing APIs in .NET Core 1.x, targeting

.NET Core 2.0 is likely to work.

If the project file contains `<RuntimeFrameworkVersion>1.{sub-version}</RuntimeFrameworkVersion>`, see this GitHub issue.

Update .NET Core SDK version in global.json

If your solution relies upon a global.json file to target a specific .NET Core SDK version, update its `version` property to use the 2.0 version installed on your machine:

JSON

```json
{
  "sdk": {
    "version": "2.0.0"
  }
}
```

Update package references

The `.csproj` file in a 1.x project lists each NuGet package used by the project.

In an ASP.NET Core 2.0 project targeting .NET Core 2.0, a single metapackage reference in the `.csproj` file replaces the collection of packages:

XML

```xml
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
</ItemGroup>
```

All the features of ASP.NET Core 2.0 and Entity Framework Core 2.0 are included in the metapackage.

ASP.NET Core 2.0 projects targeting .NET Framework should continue to reference individual NuGet packages. Update the `Version` attribute of each `<PackageReference />` node to 2.0.0.

For example, here's the list of `<PackageReference />` nodes used in a typical ASP.NET Core 2.0 project targeting .NET Framework:

XML

```xml
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Authentication.Cookies"
Version="2.0.0" />
  <PackageReference
Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore"
Version="2.0.0" />
  <PackageReference
Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="2.0.0"
/>
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation"
Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles"
Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink"
Version="2.0.0" />
  <PackageReference
Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0"
PrivateAssets="All" />
</ItemGroup>
```

The package `Microsoft.Extensions.CommandLineUtils` has been retired. It is still available but unsupported.

Update .NET CLI tools

In the `.csproj` file, update the `Version` attribute of each `<DotNetCliToolReference />` node to 2.0.0.

For example, here's the list of CLI tools used in a typical ASP.NET Core 2.0 project targeting .NET Core 2.0:

XML

```xml
<ItemGroup>
  <DotNetCliToolReference
Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools"
Version="2.0.0" />
  <DotNetCliToolReference
```

```
Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
</ItemGroup>
```

Rename Package Target Fallback property

The `.csproj` file of a 1.x project used a `PackageTargetFallback` node and variable:

XML

```xml
<PackageTargetFallback>$(PackageTargetFallback);portable-
net45+win8+wp8+wpa81;</PackageTargetFallback>
```

Rename both the node and variable to `AssetTargetFallback`:

XML

```xml
<AssetTargetFallback>$(AssetTargetFallback);portable-net45+win8+wp8+wpa81;
</AssetTargetFallback>
```

Update Main method in Program.cs

In 1.x projects, the `Main` method of `Program.cs` looked like this:

C#

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

namespace AspNetCoreDotNetCore1App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .UseApplicationInsights()
                .Build();

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

In 2.0 projects, the `Main` method of `Program.cs` has been simplified:

```C#
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore2App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

The adoption of this new 2.0 pattern is highly recommended and is required for product features like Entity Framework (EF) Core Migrations to work. For example, running `Update-Database` from the Package Manager Console window or `dotnet ef database update` from the command line (on projects converted to ASP.NET Core 2.0) generates the following error:

```
Unable to create an object of type '<Context>'. Add an implementation of
'IDesignTimeDbContextFactory<Context>' to the project, or see
https://go.microsoft.com/fwlink/?linkid=851728 for additional patterns
supported at design time.
```

Add configuration providers

In 1.x projects, adding configuration providers to an app was accomplished via the `Startup` constructor. The steps involved creating an instance of `ConfigurationBuilder`, loading applicable providers (environment variables, app settings, etc.), and initializing a member of `IConfigurationRoot`.

C#

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: false, reloadOnChange:
true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional:
true);

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets<Startup>();
    }

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

public IConfigurationRoot Configuration { get; }
```

The preceding example loads the `Configuration` member with configuration settings from `appsettings.json` as well as any `appsettings.{Environment}.json` file matching the `IHostingEnvironment.EnvironmentName` property. The location of these files is at the same path as `Startup.cs`.

In 2.0 projects, the boilerplate configuration code inherent to 1.x projects runs behind-the-scenes. For example, environment variables and app settings are loaded at startup. The equivalent `Startup.cs` code is reduced to `IConfiguration` initialization with the injected instance:

```C#
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

To remove the default providers added by `WebHostBuilder.CreateDefaultBuilder`, invoke the `Clear` method on the `IConfigurationBuilder.Sources` property inside of `ConfigureAppConfiguration`. To add providers back, utilize the `ConfigureAppConfiguration` method in `Program.cs`:

C#

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

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureAppConfiguration((hostContext, config) =>
        {
            // delete all default configuration providers
            config.Sources.Clear();
            config.AddJsonFile("myconfig.json", optional: true);
        })
        .Build();
```

The configuration used by the `CreateDefaultBuilder` method in the preceding code snippet can be seen here.

For more information, see Configuration in ASP.NET Core.

Move database initialization code

In 1.x projects using EF Core 1.x, a command such as `dotnet ef migrations add` does the following:

1. Instantiates a `Startup` instance
2. Invokes the `ConfigureServices` method to register all services with dependency injection (including `DbContext` types)
3. Performs its requisite tasks

In 2.0 projects using EF Core 2.0, `Program.BuildWebHost` is invoked to obtain the application services. Unlike 1.x, this has the additional side effect of invoking `Startup.Configure`. If your 1.x app invoked database initialization code in its `Configure` method, unexpected problems can occur. For example, if the database doesn't yet exist, the seeding code runs before the EF Core Migrations command execution. This problem causes a `dotnet ef migrations list` command to fail if the database doesn't yet exist.

Consider the following 1.x seed initialization code in the `Configure` method of `Startup.cs`:

C#

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

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

In 2.0 projects, move the `SeedData.Initialize` call to the `Main` method of `Program.cs`:

```C#
var host = BuildWebHost(args);

using (var scope = host.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    try
    {
        // Requires using RazorPagesMovie.Models;
        SeedData.Initialize(services);
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred seeding the DB.");
    }
}

host.Run();
```

As of 2.0, it's bad practice to do anything in `BuildWebHost` except build and configure the web host. Anything that's about running the application should be handled outside of `BuildWebHost` — typically in the `Main` method of `Program.cs`.

Review Razor view compilation setting

Faster application startup time and smaller published bundles are of utmost importance to you. For these reasons, Razor view compilation is enabled by default in ASP.NET Core 2.0.

Setting the `MvcRazorCompileOnPublish` property to true is no longer required. Unless you're disabling view compilation, the property may be removed from the `.csproj` file.

When targeting .NET Framework, you still need to explicitly reference the Microsoft.AspNetCore.Mvc.Razor.ViewCompilation NuGet package in your `.csproj` file:

XML

```xml
<PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation"
Version="2.0.0" PrivateAssets="All" />
```

Rely on Application Insights "light-up" features

Effortless setup of application performance instrumentation is important. You can now rely on the new Application Insights "light-up" features available in the Visual Studio 2017 tooling.

ASP.NET Core 1.1 projects created in Visual Studio 2017 added Application Insights by default. If you're not using the Application Insights SDK directly, outside of `Program.cs` and `Startup.cs`, follow these steps:

1. If targeting .NET Core, remove the following `<PackageReference />` node from the `.csproj` file:

   XML

   ```xml
   <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore"
   Version="2.0.0" />
   ```

2. If targeting .NET Core, remove the `UseApplicationInsights` extension method invocation from `Program.cs`:

   C#

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

       host.Run();
   }
   ```

3. Remove the Application Insights client-side API call from `_Layout.cshtml`. It comprises the following two lines of code:

CSHTML

```cshtml
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet
JavaScriptSnippet
@Html.Raw(JavaScriptSnippet.FullScript)
```

If you are using the Application Insights SDK directly, continue to do so. The 2.0 metapackage includes the latest version of Application Insights, so a package downgrade error appears if you're referencing an older version.

Adopt authentication/Identity improvements

ASP.NET Core 2.0 has a new authentication model and a number of significant changes to ASP.NET Core Identity. If you created your project with Individual User Accounts enabled, or if you have manually added authentication or Identity, see Migrate Authentication and Identity to ASP.NET Core 2.0.

Additional resources

- Breaking Changes in ASP.NET Core 2.0

Migrate authentication and Identity to ASP.NET Core 2.0

Article • 06/18/2024

By Scott Addie and Hao Kung

ASP.NET Core 2.0 has a new model for authentication and Identity that simplifies configuration by using services. ASP.NET Core 1.x applications that use authentication or Identity can be updated to use the new model as outlined below.

Update namespaces

In 1.x, classes such `IdentityRole` and `IdentityUser` were found in the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` namespace.

In 2.0, the Microsoft.AspNetCore.Identity namespace became the new home for several of such classes. With the default Identity code, affected classes include `ApplicationUser` and `Startup`. Adjust your `using` statements to resolve the affected references.

Authentication Middleware and services

In 1.x projects, authentication is configured via middleware. A middleware method is invoked for each authentication scheme you want to support.

The following 1.x example configures Facebook authentication with Identity in `Startup.cs`:

```C#
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>();
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory)
{
    app.UseIdentity();
    app.UseFacebookAuthentication(new FacebookOptions {
        AppId = Configuration["auth:facebook:appid"],
        AppSecret = Configuration["auth:facebook:appsecret"]
    });
}
```

In 2.0 projects, authentication is configured via services. Each authentication scheme is registered in the `ConfigureServices` method of `Startup.cs`. The `UseIdentity` method is replaced with `UseAuthentication`.

The following 2.0 example configures Facebook authentication with Identity in `Startup.cs`:

```C#
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>();

    // If you want to tweak Identity cookies, they're no longer part of
IdentityOptions.
    services.ConfigureApplicationCookie(options => options.LoginPath =
"/Account/LogIn");
    services.AddAuthentication()
            .AddFacebook(options =>
            {
                options.AppId = Configuration["auth:facebook:appid"];
                options.AppSecret =
Configuration["auth:facebook:appsecret"];
            });
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory)
{
    app.UseAuthentication();
}
```

The `UseAuthentication` method adds a single authentication middleware component, which is responsible for automatic authentication and the handling of remote authentication requests. It replaces all of the individual middleware components with a single, common middleware component.

Below are 2.0 migration instructions for each major authentication scheme.

Cookie-based authentication

Select one of the two options below, and make the necessary changes in `Startup.cs`:

1. Use cookies with Identity

- Replace `UseIdentity` with `UseAuthentication` in the `Configure` method:

```
C#
```

```csharp
app.UseAuthentication();
```

- Invoke the `AddIdentity` method in the `ConfigureServices` method to add the cookie authentication services.

- Optionally, invoke the `ConfigureApplicationCookie` or `ConfigureExternalCookie` method in the `ConfigureServices` method to tweak the Identity cookie settings.

```
C#
```

```csharp
services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

services.ConfigureApplicationCookie(options => options.LoginPath =
"/Account/LogIn");
```

2. Use cookies without Identity

- Replace the `UseCookieAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
C#
```

```csharp
app.UseAuthentication();
```

- Invoke the `AddAuthentication` and `AddCookie` methods in the `ConfigureServices` method:

```
C#
```

```csharp
// If you don't want the cookie to be automatically authenticated
and assigned to HttpContext.User,
// remove the CookieAuthenticationDefaults.AuthenticationScheme
parameter passed to AddAuthentication.
services.AddAuthentication(CookieAuthenticationDefaults.Authentica
tionScheme)
        .AddCookie(options =>
        {
            options.LoginPath = "/Account/LogIn";
            options.LogoutPath = "/Account/LogOff";
        });
```

JWT Bearer Authentication

Make the following changes in `Startup.cs`:

- Replace the `UseJwtBearerAuthentication` method call in the `Configure` method with `UseAuthentication`:

  ```C#
  app.UseAuthentication();
  ```

- Invoke the `AddJwtBearer` method in the `ConfigureServices` method:

  ```C#
  services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
          .AddJwtBearer(options =>
          {
              options.Audience = "http://localhost:5001/";
              options.Authority = "http://localhost:5000/";
          });
  ```

  This code snippet doesn't use Identity, so the default scheme should be set by passing `JwtBearerDefaults.AuthenticationScheme` to the `AddAuthentication` method.

OpenID Connect (OIDC) authentication

Make the following changes in `Startup.cs`:

- Replace the `UseOpenIdConnectAuthentication` method call in the `Configure` method with `UseAuthentication`:

  ```C#
  app.UseAuthentication();
  ```

- Invoke the `AddOpenIdConnect` method in the `ConfigureServices` method:

  ```C#
  ```

```
services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.Authority = Configuration["auth:oidc:authority"];
    options.ClientId = Configuration["auth:oidc:clientid"];
});
```

- Replace the `PostLogoutRedirectUri` property in the `OpenIdConnectOptions` action with `SignedOutRedirectUri`:

C#

```
.AddOpenIdConnect(options =>
{
    options.SignedOutRedirectUri = "https://contoso.com";
});
```

Facebook authentication

Make the following changes in `Startup.cs`:

- Replace the `UseFacebookAuthentication` method call in the `Configure` method with `UseAuthentication`:

C#

```
app.UseAuthentication();
```

- Invoke the `AddFacebook` method in the `ConfigureServices` method:

C#

```
services.AddAuthentication()
        .AddFacebook(options =>
        {
            options.AppId = Configuration["auth:facebook:appid"];
            options.AppSecret =
Configuration["auth:facebook:appsecret"];
        });
```

Google authentication

Make the following changes in `Startup.cs`:

- Replace the `UseGoogleAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
C#
```

```csharp
app.UseAuthentication();
```

- Invoke the `AddGoogle` method in the `ConfigureServices` method:

```
C#
```

```csharp
services.AddAuthentication()
        .AddGoogle(options =>
        {
            options.ClientId = Configuration["auth:google:clientid"];
            options.ClientSecret =
Configuration["auth:google:clientsecret"];
        });
```

Microsoft Account authentication

For more information on Microsoft account authentication, see this GitHub issue .

Make the following changes in `Startup.cs`:

- Replace the `UseMicrosoftAccountAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
C#
```

```csharp
app.UseAuthentication();
```

- Invoke the `AddMicrosoftAccount` method in the `ConfigureServices` method:

```
C#
```

```
services.AddAuthentication()
        .AddMicrosoftAccount(options =>
        {
            options.ClientId =
Configuration["auth:microsoft:clientid"];
            options.ClientSecret =
Configuration["auth:microsoft:clientsecret"];
        });
```

Twitter authentication

Make the following changes in `Startup.cs`:

- Replace the `UseTwitterAuthentication` method call in the `Configure` method with `UseAuthentication`:

  C#

  ```
  app.UseAuthentication();
  ```

- Invoke the `AddTwitter` method in the `ConfigureServices` method:

  C#

  ```
  services.AddAuthentication()
          .AddTwitter(options =>
          {
              options.ConsumerKey =
  Configuration["auth:twitter:consumerkey"];
              options.ConsumerSecret =
  Configuration["auth:twitter:consumersecret"];
          });
  ```

Setting default authentication schemes

In 1.x, the `AutomaticAuthenticate` and `AutomaticChallenge` properties of the AuthenticationOptions base class were intended to be set on a single authentication scheme. There was no good way to enforce this.

In 2.0, these two properties have been removed as properties on the individual `AuthenticationOptions` instance. They can be configured in the `AddAuthentication` method call within the `ConfigureServices` method of `Startup.cs`:

C#

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme
);
```

In the preceding code snippet, the default scheme is set to `CookieAuthenticationDefaults.AuthenticationScheme` ("Cookies").

Alternatively, use an overloaded version of the `AddAuthentication` method to set more than one property. In the following overloaded method example, the default scheme is set to `CookieAuthenticationDefaults.AuthenticationScheme`. The authentication scheme may alternatively be specified within your individual `[Authorize]` attributes or authorization policies.

```C#
services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
});
```

Define a default scheme in 2.0 if one of the following conditions is true:

- You want the user to be automatically signed in
- You use the `[Authorize]` attribute or authorization policies without specifying schemes

An exception to this rule is the `AddIdentity` method. This method adds cookies for you and sets the default authenticate and challenge schemes to the application cookie `IdentityConstants.ApplicationScheme`. Additionally, it sets the default sign-in scheme to the external cookie `IdentityConstants.ExternalScheme`.

Use HttpContext authentication extensions

The `IAuthenticationManager` interface is the main entry point into the 1.x authentication system. It has been replaced with a new set of `HttpContext` extension methods in the `Microsoft.AspNetCore.Authentication` namespace.

For example, 1.x projects reference an `Authentication` property:

C#

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.Authentication.SignOutAsync(_externalCookieScheme);
```

In 2.0 projects, import the `Microsoft.AspNetCore.Authentication` namespace, and delete the `Authentication` property references:

```C#
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

Windows Authentication (HTTP.sys / IISIntegration)

There are two variations of Windows authentication:

- The host only allows authenticated users. This variation isn't affected by the 2.0 changes.

- The host allows both anonymous and authenticated users. This variation is affected by the 2.0 changes. For example, the app should allow anonymous users at the IIS or HTTP.sys layer but authorize users at the controller level. In this scenario, set the default scheme in the `Startup.ConfigureServices` method.

  For Microsoft.AspNetCore.Server.IISIntegration, set the default scheme to `IISDefaults.AuthenticationScheme`:

  ```C#
  using Microsoft.AspNetCore.Server.IISIntegration;

  services.AddAuthentication(IISDefaults.AuthenticationScheme);
  ```

  For Microsoft.AspNetCore.Server.HttpSys, set the default scheme to `HttpSysDefaults.AuthenticationScheme`:

  ```C#
  using Microsoft.AspNetCore.Server.HttpSys;

  services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);
  ```

Failure to set the default scheme prevents the authorize (challenge) request from working with the following exception:

> `System.InvalidOperationException`: No authenticationScheme was specified, and there was no DefaultChallengeScheme found.

For more information, see Configure Windows Authentication in ASP.NET Core.

IdentityCookieOptions instances

A side effect of the 2.0 changes is the switch to using named options instead of cookie options instances. The ability to customize the Identity cookie scheme names is removed.

For example, 1.x projects use constructor injection to pass an `IdentityCookieOptions` parameter into `AccountController.cs` and `ManageController.cs`. The external cookie authentication scheme is accessed from the provided instance:

```C#
public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IOptions<IdentityCookieOptions> identityCookieOptions,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _externalCookieScheme =
identityCookieOptions.Value.ExternalCookieAuthenticationScheme;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger<AccountController>();
}
```

The aforementioned constructor injection becomes unnecessary in 2.0 projects, and the `_externalCookieScheme` field can be deleted:

```C#
public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger<AccountController>();
}
```

1.x projects used the `_externalCookieScheme` field as follows:

C#

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.Authentication.SignOutAsync(_externalCookieScheme);
```

In 2.0 projects, replace the preceding code with the following. The `IdentityConstants.ExternalScheme` constant can be used directly.

C#

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

Resolve the newly added `SignOutAsync` call by importing the following namespace:

C#

```
using Microsoft.AspNetCore.Authentication;
```

Add IdentityUser POCO navigation properties

The Entity Framework (EF) Core navigation properties of the base `IdentityUser` POCO (Plain Old CLR Object) have been removed. If your 1.x project used these properties, manually add them back to the 2.0 project:

C#

```
/// <summary>
/// Navigation property for the roles this user belongs to.
/// </summary>
public virtual ICollection<IdentityUserRole<int>> Roles { get; } = new
List<IdentityUserRole<int>>();

/// <summary>
/// Navigation property for the claims this user possesses.
/// </summary>
public virtual ICollection<IdentityUserClaim<int>> Claims { get; } = new
List<IdentityUserClaim<int>>();

/// <summary>
/// Navigation property for this users login accounts.
/// </summary>
public virtual ICollection<IdentityUserLogin<int>> Logins { get; } = new
List<IdentityUserLogin<int>>();
```

To prevent duplicate foreign keys when running EF Core Migrations, add the following to your `IdentityDbContext` class' `OnModelCreating` method (after the `base.OnModelCreating();` call):

```C#
protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);
    // Customize the ASP.NET Core Identity model and override the defaults
if needed.
    // For example, you can rename the ASP.NET Core Identity table names and
more.
    // Add your customizations after calling base.OnModelCreating(builder);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Claims)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Logins)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Roles)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);
}
```

Replace GetExternalAuthenticationSchemes

The synchronous method `GetExternalAuthenticationSchemes` was removed in favor of an asynchronous version. 1.x projects have the following code in `Controllers/ManageController.cs`:

```C#
var otherLogins =
_signInManager.GetExternalAuthenticationSchemes().Where(auth =>
userLogins.All(ul => auth.AuthenticationScheme !=
ul.LoginProvider)).ToList();
```

This method appears in `Views/Account/Login.cshtml` too:

```CSHTML
@{
    var loginProviders =
SignInManager.GetExternalAuthenticationSchemes().ToList();
    if (loginProviders.Count == 0)
    {
        <div>
            <p>
                There are no external authentication services configured.
See <a href="https://go.microsoft.com/fwlink/?LinkID=532715">this
article</a>
                for details on setting up this ASP.NET application to
support logging in via external services.
            </p>
        </div>
    }
    else
    {
        <form asp-controller="Account" asp-action="ExternalLogin" asp-route-
returnurl="@ViewData["ReturnUrl"]" method="post" class="form-horizontal">
            <div>
                <p>
                    @foreach (var provider in loginProviders)
                    {
                        <button type="submit" class="btn btn-default"
name="provider" value="@provider.AuthenticationScheme" title="Log in using
your @provider.DisplayName account">@provider.AuthenticationScheme</button>
                    }
                </p>
            </div>
        </form>
    }
}
```

In 2.0 projects, use the GetExternalAuthenticationSchemesAsync method. The change in `ManageController.cs` resembles the following code:

C#

```csharp
var schemes = await _signInManager.GetExternalAuthenticationSchemesAsync();
var otherLogins = schemes.Where(auth => userLogins.All(ul => auth.Name !=
ul.LoginProvider)).ToList();
```

In `Login.cshtml`, the `AuthenticationScheme` property accessed in the `foreach` loop changes to `Name`:

CSHTML

```cshtml
@{
    var loginProviders = (await
SignInManager.GetExternalAuthenticationSchemesAsync()).ToList();
    if (loginProviders.Count == 0)
    {
        <div>
            <p>
                There are no external authentication services configured.
See <a href="https://go.microsoft.com/fwlink/?LinkID=532715">this
article</a>
                for details on setting up this ASP.NET application to
support logging in via external services.
            </p>
        </div>
    }
    else
    {
        <form asp-controller="Account" asp-action="ExternalLogin" asp-route-
returnurl="@ViewData["ReturnUrl"]" method="post" class="form-horizontal">
            <div>
                <p>
                    @foreach (var provider in loginProviders)
                    {
                        <button type="submit" class="btn btn-default"
name="provider" value="@provider.Name" title="Log in using your
@provider.DisplayName account">@provider.DisplayName</button>
                    }
                </p>
            </div>
        </form>
    }
}
```

ManageLoginsViewModel property change

A `ManageLoginsViewModel` object is used in the `ManageLogins` action of `ManageController.cs`. In 1.x projects, the object's `OtherLogins` property return type is

`IList<AuthenticationDescription>`. This return type requires an import of `Microsoft.AspNetCore.Http.Authentication`:

```C#
using System.Collections.Generic;
using Microsoft.AspNetCore.Http.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore1App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationDescription> OtherLogins { get; set; }
    }
}
```

In 2.0 projects, the return type changes to `IList<AuthenticationScheme>`. This new return type requires replacing the `Microsoft.AspNetCore.Http.Authentication` import with a `Microsoft.AspNetCore.Authentication` import.

```C#
using System.Collections.Generic;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore2App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationScheme> OtherLogins { get; set; }
    }
}
```

Additional resources

For more information, see the Discussion for Auth 2.0 issue on GitHub.

Upgrade from ASP.NET Framework to ASP.NET Core

Article • 04/13/2023

Why upgrade to the latest .NET

ASP.NET Core is the modern web framework for .NET. While ASP.NET Core has many similarities to ASP.NET in the .NET Framework, it's a new framework that's completely rewritten. ASP.NET apps updated to ASP.NET Core can benefit from improved performance and access to the latest web development features and capabilities.

ASP.NET Framework update approaches

Most non-trivial ASP.NET Framework apps should consider using the incremental upgrade approach. For more information, see Incremental ASP.NET to ASP.NET Core upgrade.

For ASP.NET MVC and Web API apps, see Learn to upgrade from ASP.NET MVC and Web API to ASP.NET Core MVC. For ASP.NET Framework Web Forms apps, see Learn to upgrade from ASP.NET Web Forms to ASP.NET Core.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

URI decoding differences between ASP.NET to ASP.NET Core

ASP.NET Core has the following URI decoding differences with ASP.NET Framework:

Expand table

ASCII	Encoded	ASP.NET Core	ASP.NET Framework
`\`	`%5C`	`\`	`/`

ASCII	Encoded	ASP.NET Core	ASP.NET Framework
`/`	`%2F`	`%2F`	`/`

When decoding `%2F` on ASP.NET Core:

- The entire path gets unescaped except `%2F` because converting it to `/` would change the path structure. It can’t be decoded until the path is split into segments.

To generate the value for `HttpRequest.Url`, use `new Uri(this.AspNetCoreHttpRequest.GetEncodedUrl());` to avoid `Uri` misinterpreting the values.

Migrating User Secrets from ASP.NET Framework to ASP.NET Core

See this GitHub issue.

Incremental ASP.NET to ASP.NET Core update

Article • 01/11/2024

Updating an app from ASP.NET Framework to ASP.NET Core is non-trivial for the majority of production apps. These apps often incorporate new technologies as they become available and are often composed of many legacy decisions. This article provides guidance and links to tools for updating ASP.NET Framework apps to ASP.NET Core with as little change as possible.

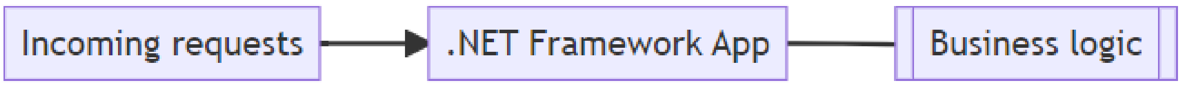
One of the larger challenges is the pervasive use of HttpContext throughout a code base. Without the incremental approach and tools, a large scale rewrite is required to remove the HttpContext dependency. The adapters in dotnet/systemweb-adapters provide a set of runtime helpers to access the types used in the ASP.NET Framework app in a way that works in ASP.NET Core with minimal changes.

A complete migration may take considerable effort depending on the size of the app, dependencies, and non-portable APIs used. In order to keep deploying an app to production while working on updating, the best pattern is to follow is the Strangler Fig pattern. The *Strangler Fig pattern* allows for continual development on the old system with an incremental approach to replacing specific pieces of functionality with new services. This document describes how to apply the Strangler Fig pattern to an ASP.NET app updating towards ASP.NET Core.

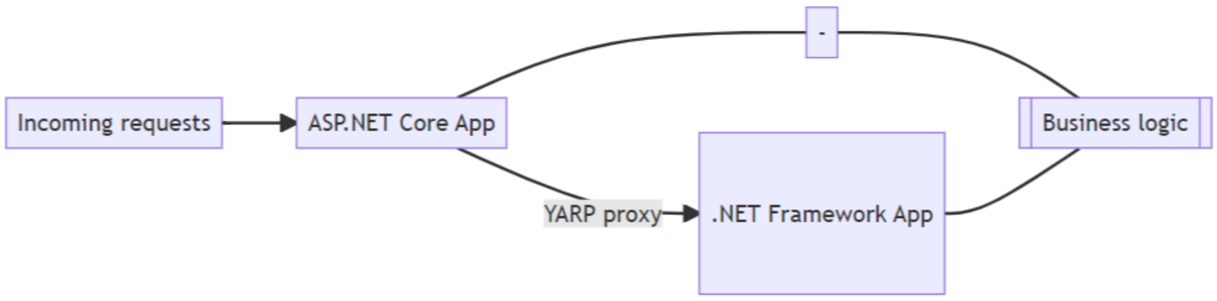
If you'd like to skip this overview article and get started, see Get started.

App migration to ASP.NET Core

Before starting the migration, the app targets ASP.NET Framework and runs on Windows with its supporting libraries:

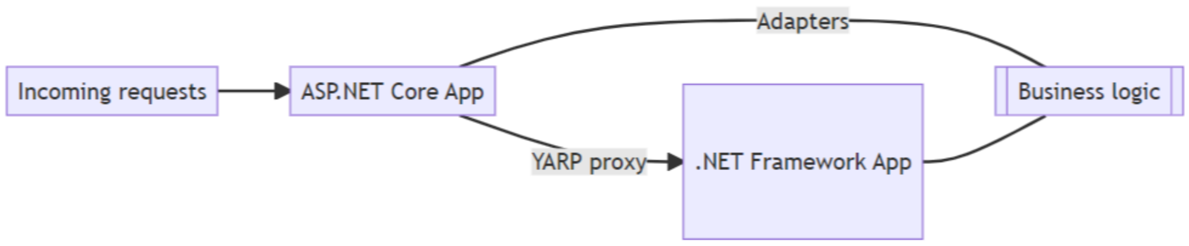


Migration starts by introducing a new app based on ASP.NET Core that becomes the entry point. Incoming requests go to the ASP.NET Core app, which either handles the request or proxies the request to the .NET Framework app via YARP. At first, the majority of code providing responses is in the .NET Framework app, but the ASP.NET Core app is now set up to start migrating routes:



To migrate business logic that relies on `HttpContext`, the libraries need to be built with `Microsoft.AspNetCore.SystemWebAdapters`. Building the libraries with `SystemWebAdapters` allows:

- The libraries to be built against .NET Framework, .NET Core, or .NET Standard 2.0.
- Ensures that the libraries are using APIs that are available on both ASP.NET Framework and ASP.NET Core.

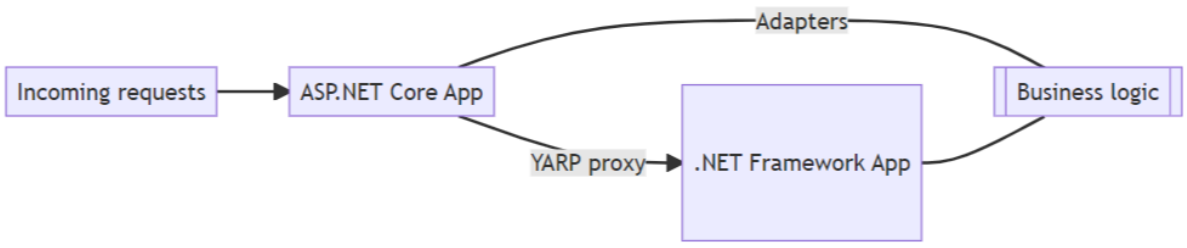


Once the ASP.NET Core app using YARP is set up, you can start updating routes from ASP.NET Framework to ASP.NET Core. For example, WebAPI or MVC controller action methods,handlers, or some other implementation of a route. If the route is available in the ASP.NET Core app, it's matched and served.

During the migration process, additional services and infrastructure are identified that must be updated to run on .NET Core. Options listed in order of maintainability include:

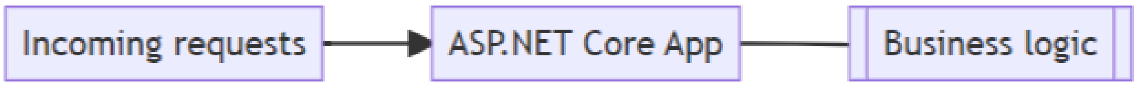
1. Move the code to shared libraries
2. Link the code in the new project
3. Duplicate the code

Eventually, the ASP.NET Core app handles more of the routes than the .NET Framework app:



Once the ASP.NET Framework app is no longer needed and deleted:

- The app is running on the ASP.NET Core app stack, but is still using the adapters.
- The remaining migration work is removing the use of adapters.



The Visual Studio extension .NET Upgrade Assistant can help upgrade ASP.NET Framework web apps to ASP.NET Core. For more information see the blog post Upgrading your .NET projects with Visual Studio.

System.Web Adapters

The `Microsoft.AspNetCore.SystemWebAdapters` namespace is a collection of runtime helpers that facilitate using code written against `System.Web` while moving to ASP.NET Core. There are a few packages that may be used to use features from these adapters:

- `Microsoft.AspNetCore.SystemWebAdapters`: This package is used in supporting libraries and provide the System.Web APIs you may have taken a dependency on, such as `HttpContext` and others. This package targets .NET Standard 2.0, .NET 4.5+, and .NET 6+.
- `Microsoft.AspNetCore.SystemWebAdapters.FrameworkServices`: This package only targets .NET Framework and is intended to provide services to ASP.NET Framework applications that may need to provide incremental migrations. This is generally not expected to be referenced from libraries, but rather from the applications themselves.
- `Microsoft.AspNetCore.SystemWebAdapters.CoreServices`: This package only targets .NET 6+ and is intended to provide services to ASP.NET Core applications to configure behavior of `System.Web` APIs as well as opting into any behaviors for incremental migration. This is generally not expected to be referenced from libraries, but rather from the applications themselves.

- `Microsoft.AspNetCore.SystemWebAdapters.Abstractions`: This package is a supporting package that provides abstractions for services used by both the ASP.NET Core and ASP.NET Framework application such as session state serialization.

For examples of scenarios where this is useful, see the adapters article.

For guidance around usage, see the usage guidance article.

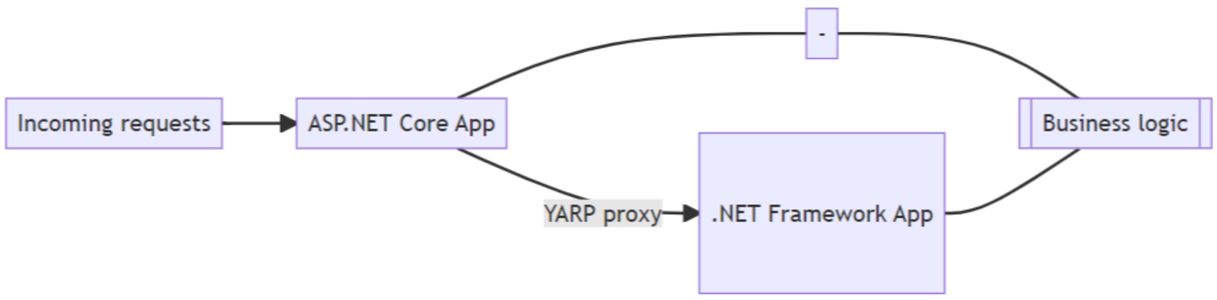
Additional Resources

- Example migration of eShop to ASP.NET Core
- Video:Tooling for Incremental ASP.NET Core Migrations
- Unit Testing

Get started with incremental ASP.NET to ASP.NET Core migration

Article • 01/11/2024

For a large migration, we recommend setting up a ASP.NET Core app that proxies to the original .NET Framework app. The new proxy enabled app is shown in the following image:



To understand how this approach is helpful in the migration process, see Incremental ASP.NET to ASP.NET Core migration. The rest of this article provides the steps to proceed with an incremental migration.

Set up ASP.NET Core Project

For ASP.NET MVC and Web API apps, see Learn to upgrade from ASP.NET MVC and Web API to ASP.NET Core MVC. For ASP.NET Framework Web Forms apps, see Learn to upgrade from ASP.NET Web Forms to ASP.NET Core.

Upgrade supporting libraries

If you have supporting libraries in your solution that you will need to use, they should be upgraded to .NET Standard 2.0, if possible. Upgrade Assistant is a great tool for this. If libraries are unable to target .NET Standard, you can target .NET 6 or later either along with the .NET Framework target in the original project or in a new project alongside the original.

The adapters can be used in these libraries to enable support for `System.Web.HttpContext` usage in class libraries. In order to enable `System.Web.HttpContext` usage in a library:

1. Remove reference to `System.Web` in the project file

2. Add the `Microsoft.AspNetCore.SystemWebAdapters` package
3. Enable multi-targeting and add a .NET 6 target or later, or convert the project to .NET Standard 2.0.
4. Ensure the target framework supports .NET Core. Multi-targeting can be used if .NET Standard 2.0 is not sufficient

This step may require a number of projects to change depending on your solution structure. Upgrade Assistant can help you identify which ones need to change and automate a number of steps in the process.

Enable Session Support

Session is a commonly used feature of ASP.NET that shares the name with a feature in ASP.NET Core the APIs are much different. See the documentation on session support.

Enable shared authentication support

It is possible to share authentication between the original ASP.NET app and the new ASP.NET Core app by using the `System.Web` adapters remote authentication feature. This feature allows the ASP.NET Core app to defer authentication to the ASP.NET app. See the remote app connection and remote authentication docs for more details.

General Usage Guidance

There are a number of differences between ASP.NET and ASP.NET Core that the adapters are able to help update. However, there are some features that require an opt-in as they incur some cost. There are also behaviors that cannot be adapted. See usage guidance for a list of these.

System.Web adapters

Article • 01/11/2024

The main use case of the adapters in the dotnet/systemweb-adapters repository is to help developers who have taken a reliance on `System.Web` types within their class libraries as they want to move to ASP.NET Core.

An important feature of the adapters is the adapters allow the library to be used from ***both*** ASP.NET Framework and ASP.NET Core projects. Updating multiple ASP.NET Framework apps to ASP.NET Core often involves intermediate states where not all the apps have been fully updated. By using the `System.Web` adapters, the library can be used both from ASP.NET Core callers and ASP.NET Framework callers that haven't been upgraded.

Let's take a look at an example using the adapters moving from .NET Framework to ASP.NET Core.

ASP.NET Framework

Consider a controller that does something such as:

```cs
public class SomeController : Controller
{
  public ActionResult Index()
  {
    SomeOtherClass.SomeMethod(HttpContext.Current);
  }
}
```

which then has logic in a separate assembly passing that HttpContext around until finally, some inner method does some logic on it such as:

```cs
public class Class2
{
  public bool PerformSomeCheck(HttpContext context)
  {
    return context.Request.Headers["SomeHeader"] == "ExpectedValue";
  }
}
```

ASP.NET Core

In order to run the above logic in ASP.NET Core, a developer will need to add the `Microsoft.AspNetCore.SystemWebAdapters` package, that will enable the projects to work on both platforms.

The libraries would need to be updated to understand the adapters, but it will be as simple as adding the package and recompiling. If these are the only dependencies a system has on `System.Web.dll`, then the libraries will be able to target .NET Standard 2.0 to facilitate a simpler building process while migrating.

The controller in ASP.NET Core will now look like this:

```cs
public class SomeController : Controller
{
  [Route("/")]
  public IActionResult Index()
  {
    SomeOtherClass.SomeMethod(HttpContext);
  }
}
```

Notice that since there's a HttpContext property, they can pass that through, but it generally looks the same. Using implicit conversions, the HttpContext can be converted into the adapter that could then be passed around through the levels utilizing the code in the same way.

See also

- Unit Testing

Remote app setup

Article • 01/11/2024

In some incremental upgrade scenarios, it's useful for the new ASP.NET Core app to be able to communicate with the original ASP.NET app.

Specifically, this capability is used, currently, for remote app authentication and remote session features.

Configuration

To enable the ASP.NET Core app to communicate with the ASP.NET app, it's necessary to make a couple small changes to each app.

ASP.NET app configuration

To set up the ASP.NET app to be able to receive requests from the ASP.NET Core app:

1. Install the nuget package Microsoft.AspNetCore.SystemWebAdapters.FrameworkServices
2. Call the `AddRemoteAppServer` extension method on the `ISystemWebAdapterBuilder`:

CSharp

```csharp
SystemWebAdapterConfiguration.AddSystemWebAdapters(this)
    .AddRemoteAppServer(options =>
    {
        // ApiKey is a string representing a GUID
        options.ApiKey =
ConfigurationManager.AppSettings["RemoteAppApiKey"];
    });
```

In the options configuration method passed to the `AddRemoteAppServer` call, an API key must be specified. The API key is:

- Used to secure the endpoint so that only trusted callers can make requests to it.
- The same API key provided to the ASP.NET Core app when it is configured.
- A string and must be parsable as a GUID. Hyphens in the key are optional.

3. **Optional** : Add the `SystemWebAdapterModule` module to the `web.config` if it wasn't already added by NuGet. The `SystemWebAdapterModule` module is not added automatically when using SDK style projects for ASP.NET Core.

diff

```diff
  <system.webServer>
    <modules>
+      <remove name="SystemWebAdapterModule" />
+      <add name="SystemWebAdapterModule"
type="Microsoft.AspNetCore.SystemWebAdapters.SystemWebAdapterModule,
Microsoft.AspNetCore.SystemWebAdapters.FrameworkServices"
preCondition="managedHandler" />
    </modules>
</system.webServer>
```

ASP.NET Core app

To set up the ASP.NET Core app to be able to send requests to the ASP.NET app, you need to make a similar change, calling `AddRemoteApp` after registering System.Web adapter services with `AddSystemWebAdapters`.

CSharp

```csharp
builder.Services.AddSystemWebAdapters()
    .AddRemoteAppClient(options =>
    {
        options.RemoteAppUrl =
new(builder.Configuration["ReverseProxy:Clusters:fallbackCluster:Destination
s:fallbackApp:Address"]);
        options.ApiKey = builder.Configuration["RemoteAppApiKey"];
    });
```

In the preceding code:

- The `AddRemoteApp` call is used to configure the remote app's URL and the shared secret API key.
- The `RemoteAppUrl` property specifies the URL of the ASP.NET Framework app that the ASP.NET Core app communicates with. In this example, the URL is read from an existing configuration setting used by the YARP proxy that proxies requests to the ASP.NET Framework app as part of the incremental migration's *strangler fig pattern*.

With both the ASP.NET and ASP.NET Core app updated, extension methods can now be used to set up remote app authentication or remote session, as needed.

Securing the remote app connection

Because remote app features involve serving requests on new endpoints from the ASP.NET app, it's important that communication to and from the ASP.NET app be secure.

First, make sure that the API key string used to authenticate the ASP.NET Core app with the ASP.NET app is unique and kept secret. It is a best practice to not store the API key in source control. Instead, load it at runtime from a secure source such as Azure Key Vault or other secure runtime configuration. In order to encourage secure API keys, remote app connections require that the keys be non-empty GUIDs (128-bit hex numbers).

Second, because it's important for the ASP.NET Core app to be able to trust that it is requesting information from the correct ASP.NET app, the ASP.NET app should use HTTPS in any production scenarios so that the ASP.NET Core app can know responses are being served by a trusted source.

Usage Guidance

Article • 08/30/2024

`Microsoft.AspNetCore.SystemWebAdapters` provides an emulation layer to mimic behavior from ASP.NET framework on ASP.NET Core. Below are some guidelines for some of the considerations when using them:

`HttpContext` lifetime

The adapters are backed by HttpContext which cannot be used past the lifetime of a request. Thus, HttpContext when run on ASP.NET Core cannot be used past a request as well, while on ASP.NET Framework it would work at times. An ObjectDisposedException will be thrown in cases where it is used past a request end.

Recommendation: Store the values needed into a POCO and hold onto that.

Conversion to HttpContext

There are two ways to convert an HttpContext to a HttpContext:

- Implicit casting
- Constructor usage

Recommendation: For the most cases, implicit casting should be preferred as this will cache the created instance and ensure only a single HttpContext per request.

CurrentCulture is not set by default

In ASP.NET Framework, CurrentCulture was set for a request, but this is not done automatically in ASP.NET Core. Instead, you must add the appropriate middleware to your pipeline.

Recommendation: See ASP.NET Core Localization for details on how to enable this.

Simplest way to enable this with similar behavior as ASP.NET Framework would be to add the following to your pipeline:

C#

```
app.UseRequestLocalization();
```

CurrentPrincipal

In ASP.NET Framework, CurrentPrincipal and Current would be set to the current user. This is not available on ASP.NET Core out of the box. Support for this is available with these adapters by adding the `ISetThreadCurrentPrincipal` to the endpoint (available to controllers via the `SetThreadCurrentPrincipalAttribute`). However, it should only be used if the code cannot be refactored to remove usage.

Recommendation: If possible, use the property User or User instead by passing it through to the call site. If not possible, enable setting the current user and also consider setting the request to be a logical single thread (see below for details).

Request thread does not exist in ASP.NET Core

In ASP.NET Framework, a request had thread-affinity and Current would only be available if on that thread. ASP.NET Core does not have this guarantee so Current will be available within the same async context, but no guarantees about threads are made.

Recommendation: If reading/writing to the HttpContext, you must ensure you are doing so in a single-threaded way. You can force a request to never run concurrently on any async context by setting the `ISingleThreadedRequestMetadata`. This will have performance implications and should only be used if you can't refactor usage to ensure non-concurrent access. There is an implementation available to add to controllers with `SingleThreadedRequestAttribute`:

C#

```csharp
[SingleThreadedRequest]
public class SomeController : Controller
{
    ...
}
```

Request may need to be prebuffered

By default, the incoming request is not always seekable nor fully available. In order to get behavior seen in .NET Framework, you can opt into prebuffering the input stream. This will fully read the incoming stream and buffer it to memory or disk (depending on settings).

Recommendation: This can be enabled by applying endpoint metadata that implements the `IPreBufferRequestStreamMetadata` interface. This is available as an attribute

`PreBufferRequestStreamAttribute` that can be applied to controllers or methods.

To enable this on all MVC endpoints, there is an extension method that can be used as follows:

```cs
app.MapDefaultControllerRoute()
    .PreBufferRequestStream();
```

Response may require buffering

Some APIs on Response require that the output stream is buffered, such as Output, End(), Clear(), and SuppressContent.

Recommendation: In order to support behavior for Response that requires buffering the response before sending, endpoints must opt-into it with endpoint metadata implementing `IBufferResponseStreamMetadata`.

To enable this on all MVC endpoints, there is an extension method that can be used as follows:

```cs
app.MapDefaultControllerRoute()
    .BufferResponseStream();
```

Shared session state

In order to support Session, endpoints must opt-into it via metadata implementing `ISessionMetadata`.

Recommendation: To enable this on all MVC endpoints, there is an extension method that can be used as follows:

```cs
app.MapDefaultControllerRoute()
    .RequireSystemWebAdapterSession();
```

This also requires some implementation of a session store. For details of options here, see here.

Remote session exposes additional endpoint for application

The remote session support exposes an endpoint that allows the core app to retrieve session information. This may cause a potentially long-lived request to exist between the core app and the framework app, but will time out with the current request or the session timeout (by default is 20 minutes).

Recommendation: Ensure the API key used is a strong one and that the connection with the framework app is done over SSL.

Virtual directories must be identical for framework and core applications

The virtual directory setup is used for route generation, authorization, and other services within the system. At this point, no reliable method has been found to enable different virtual directories due to how ASP.NET Framework works.

Recommendation: Ensure your two applications are on different sites (hosts and/or ports) with the same application/virtual directory layout.

Wrapped ASP.NET Core session state

Article • 01/11/2024

This implementation wraps the session provided on ASP.NET Core so that it can be used with the adapters. The session will be using the same backing store as `Microsoft.AspNetCore.Http.ISession` but will provide strongly-typed access to its members.

Configuration for ASP.NET Core would look similar to the following:

```csharp
builder.Services.AddSystemWebAdapters()
    .AddJsonSessionSerializer(options =>
    {
        // Serialization/deserialization requires each session key to be registered to a type
        options.RegisterKey<int>("test-value");
        options.RegisterKey<SessionDemoModel>("SampleSessionItem");
    })
    .WrapAspNetCoreSession();
```

The framework app would not need any changes to enable this behavior.

ASP.NET to ASP.NET Core incremental session state migration

Article • 01/11/2024

Session State

Session state in ASP.NET Framework provided a number of features that ASP.NET Core does not provide. In order to update from ASP.NET Framework to Core, the adapters provide mechanisms to enable populating session state with similar behavior as `System.Web` did. Some of the differences between framework and core are:

- ASP.NET Framework would lock session usage within a session, so subsequent requests in a session are handled in a serial fashion. This is different than ASP.NET Core that does not provide any of these guarantees.
- ASP.NET Framework would serialize and deserialize objects automatically (unless being done in-memory). ASP.NET Core provides a mechanism to store a `byte[]` given a key. Any object serialization/deserialization has to be done manually by the user.

The adapter infrastructure exposes two interfaces that can be used to implement any session storage system. These are:

- `Microsoft.AspNetCore.SystemWebAdapters.ISessionManager`: This has a single method that gets passed an HttpContext and the session metadata and expects an `ISessionState` object to be returned.
- `Microsoft.AspNetCore.SystemWebAdapters.ISessionState`: This describes the state of a session object. It is used as the backing of the HttpSessionState type.

Serialization

Since the adapters provide the ability to work with strongly-typed session state, we must be able to serialize and deserialize types. This is customized through the `Microsoft.AspNetCore.SystemWebAdapters.SessionState.Serialization.ISessionKeySerializer`.

A default JSON implementation is provided that is configured via the `JsonSessionSerializerOptions`:

- `RegisterKey<T>(string)` - Registers a session key to a known type. This is required in order to serialize/deserialize the session state correctly. If a key is found that there is no registration for, an error will be thrown and session will not be available.

C#

```csharp
builder.Services.AddSystemWebAdapters()
    .AddJsonSessionSerializer(options =>
    {
        // Serialization/deserialization requires each session key to be
registered to a type
        options.RegisterKey<int>("test-value");
    });
```

Implementations

There are two available implementations of the session state object that currently ship, each with some trade offs of features. The best choice for an app may depend on which part of the migration it is in, and may change over time.

- Strongly typed: Provides the ability to access an object and can be cast to the expected type
- Locking: Ensures multiple requests within a single session are queued up and aren't accessing the session at the same time
- Standalone: Use when you're not sharing session between ASP.NET Framework and ASP.NET Core to avoid modifying code in class libraries that references SessionState

Below are the available implementations:

Expand table

Implementation	Strongly typed	Locking	Standalone
Remote app	✓	✓	⛔
Wrapped ASP.NET Core	✓	⛔	✓

ASP.NET to ASP.NET Core incremental IHttpModule migration

Article • 03/23/2024

Modules are types that implement IHttpModule and are used in ASP.NET Framework to hook into the request pipeline at various events. In an ASP.NET Core application, these should ideally be migrated to middleware. However, there are times when this cannot be done. In order to support migration scenarios in which modules are required and cannot be moved to middleware, System.Web adapters support adding them to ASP.NET Core.

IHttpModule Example

In order to support modules, an instance of HttpApplication must be available. If no custom HttpApplication is used, a default one will be used to add the modules to. Events declared in a custom application (including `Application_Start`) will be registered and run accordingly.

```C#
using System.Web;
using Microsoft.AspNetCore.OutputCaching;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSystemWebAdapters()
    .AddHttpApplication<MyApp>(options =>
    {
        // Size of pool for HttpApplication instances. Should be what the
expected concurrent requests will be
        options.PoolSize = 10;

        // Register a module (optionally) by name
        options.RegisterModule<MyModule>("MyModule");
    });

// Only available in .NET 7+
builder.Services.AddOutputCache(options =>
{
    options.AddHttpApplicationBasePolicy(_ => new[] { "browser" });
});

builder.Services.AddAuthentication();
builder.Services.AddAuthorization();

var app = builder.Build();
```

```
app.UseAuthentication();
app.UseAuthenticationEvents();

app.UseAuthorization();
app.UseAuthorizationEvents();

app.UseSystemWebAdapters();
app.UseOutputCache();

app.MapGet("/", () => "Hello World!")
    .CacheOutput();

app.Run();

class MyApp : HttpApplication
{
    protected void Application_Start()
    {
    }

    public override string? GetVaryByCustomString(System.Web.HttpContext
context, string custom)
    {
        // Any custom vary-by string needed

        return base.GetVaryByCustomString(context, custom);
    }
}

class MyModule : IHttpModule
{
    public void Init(HttpApplication application)
    {
        application.BeginRequest += (s, e) =>
        {
            // Handle events at the beginning of a request
        };

        application.AuthorizeRequest += (s, e) =>
        {
            // Handle events that need to be authorized
        };
    }

    public void Dispose()
    {
    }
}
```

Global.asax migration

This infrastructure can be used to migrate usage of `Global.asax` if needed. The source from `Global.asax` is a custom HttpApplication and the file can be included in an ASP.NET Core application. Since it is named `Global`, the following code can be used to register it:

```C#
builder.Services.AddSystemWebAdapters()
    .AddHttpApplication<Global>();
```

As long as the logic within it is available in ASP.NET Core, this approach can be used to incrementally migrate reliance on `Global.asax` to ASP.NET Core.

Authentication/Authorization events

In order for the authentication and authorization events to run at the desired time, the following pattern should be used:

```C#
app.UseAuthentication();
app.UseAuthenticationEvents();

app.UseAuthorization();
app.UseAuthorizationEvents();
```

If this is not done, the events will still run. However, it will be during the call of `.UseSystemWebAdapters()`.

HTTP Module pooling

Because modules and applications in ASP.NET Framework were assigned to a request, a new instance is needed for each request. However, since they can be expensive to create, they are pooled using ObjectPool<T>. In order to customize the actual lifetime of the HttpApplication instances, a custom pool can be used:

```C#
builder.Services.TryAddSingleton<ObjectPool<HttpApplication>>(sp =>
{
    // Recommended to use the in-built policy as that will ensure everything
is initialized correctly and is not intended to be replaced
    var policy = sp.GetRequiredService<IPooledObjectPolicy<HttpApplication>>
();
```

```
    // Can use any provider needed
    var provider = new DefaultObjectPoolProvider();

    // Use the provider to create a custom pool that will then be used for
the application.
    return provider.Create(policy);
});
```

Additional resources

- HTTP Module Migration
- HTTP Handlers and HTTP Modules Overview

Remote app session state

Article • 01/11/2024

Remote app session state will enable communication between the ASP.NET Core and ASP.NET app to retrieve the session state. This is enabled by exposing an endpoint on the ASP.NET app that can be queried to retrieve and set the session state.

HttpSessionState serialization

The HttpSessionState object must be serialized for remote app session state to be enabled. This is accomplished through implementation of the type `Microsoft.AspNetCore.SystemWebAdapters.SessionState.Serialization.ISessionSerializer`, of which a default binary writer implementation is provided. This is added by the following code:

C#

```csharp
builder.Services.AddSystemWebAdapters()
    .AddSessionSerializer(options =>
    {
        // Customize session serialization here
    });
```

Configuration

First, follow the remote app setup instructions to connect the ASP.NET Core and ASP.NET apps. Then, there are just a couple extra extension methods to call to enable remote app session state.

Configuration for ASP.NET Core involves calling `AddRemoteAppSession` and `AddJsonSessionSerializer` to register known session item types. The code should look similar to the following:

C#

```csharp
builder.Services.AddSystemWebAdapters()
    .AddJsonSessionSerializer(options =>
    {
        // Serialization/deserialization requires each session key to be
registered to a type
        options.RegisterKey<int>("test-value");
        options.RegisterKey<SessionDemoModel>("SampleSessionItem");
    })
    .AddRemoteAppClient(options =>
    {
        // Provide the URL for the remote app that has enabled session
querying
        options.RemoteAppUrl =
new(builder.Configuration["ReverseProxy:Clusters:fallbackCluster:Destination
s:fallbackApp:Address"]);

        // Provide a strong API key that will be used to authenticate the
request on the remote app for querying the session
        options.ApiKey = builder.Configuration["RemoteAppApiKey"];
    })
    .AddSessionClient();
```

Session support requires additional work for the ASP.NET Core pipeline, and is not turned on by default. It can be configured on a per-route basis via ASP.NET Core metadata.

For example, session support requires either to annotate a controller:

```C#
[Session]
public class SomeController : Controller
{
}
```

or to enable for all endpoints by default:

```C#
app.MapDefaultControllerRoute()
    .RequireSystemWebAdapterSession();
```

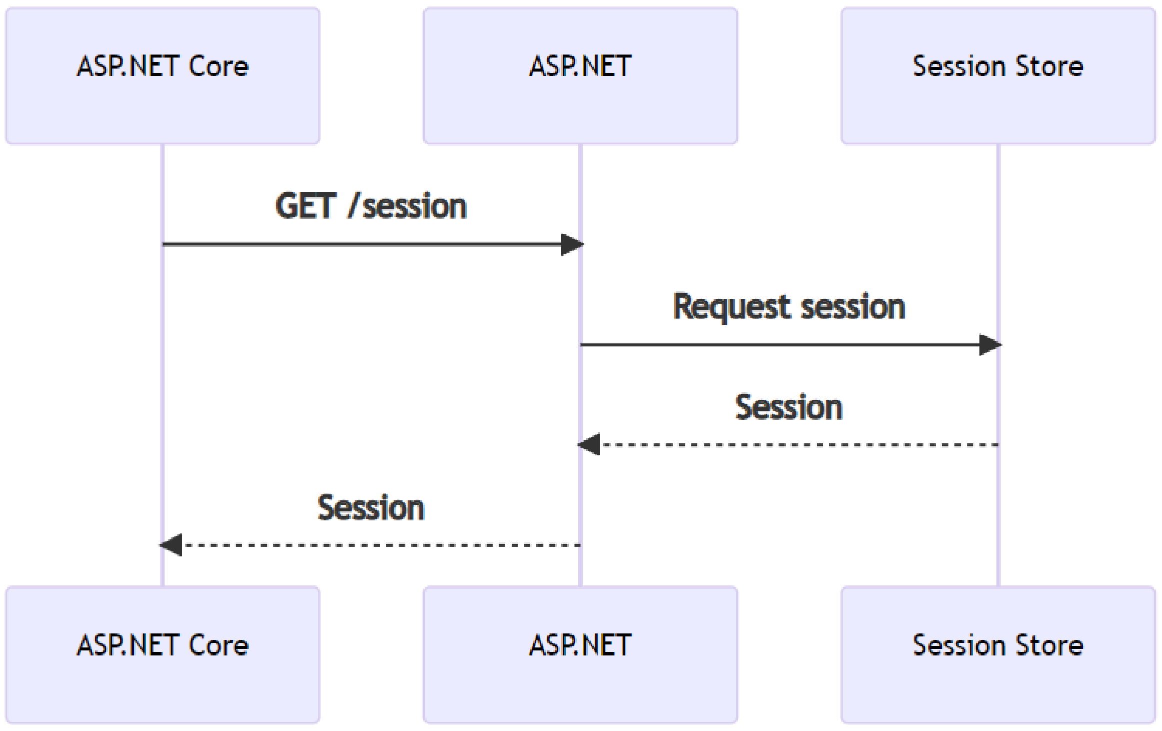
The framework equivalent would look like the following change in `Global.asax.cs`:

```C#
SystemWebAdapterConfiguration.AddSystemWebAdapters(this)
    .AddJsonSessionSerializer(options =>
    {
        // Serialization/deserialization requires each session key to be
registered to a type
        options.RegisterKey<int>("test-value");
        options.RegisterKey<SessionDemoModel>("SampleSessionItem");
    })
    // Provide a strong API key that will be used to authenticate the
request on the remote app for querying the session
    // ApiKey is a string representing a GUID
    .AddRemoteAppServer(options => options.ApiKey =
ConfigurationManager.AppSettings["RemoteAppApiKey"])
    .AddSessionServer();
```

Protocol

Readonly

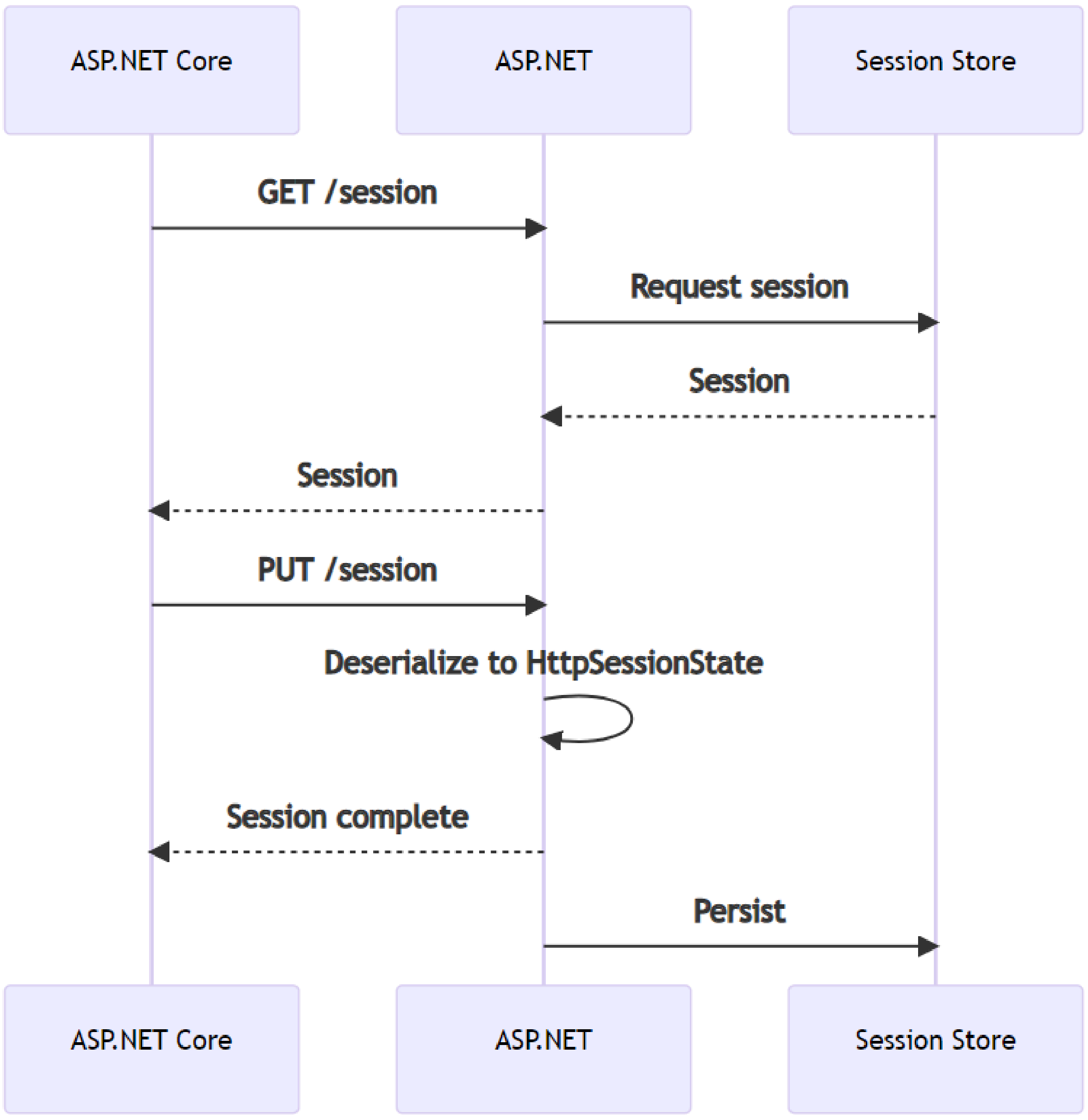
Readonly session will retrieve the session state from the framework app without any sort of locking. This consists of a single `GET` request that will return a session state and can be closed immediately.



Writeable

Writeable session state protocol starts with the same as the readonly, but differs in the following:

- Requires an additional `PUT` request to update the state
- The initial `GET` request must be kept open until the session is done; if closed, the session will not be able to be updated


ASP.NET Core
ASP.NET
Session Store
GET /session
Request session
Session
Session
PUT /session
Deserialize to HttpSessionState
Session complete
Persist
ASP.NET Core
ASP.NET
Session Store

Remote Authentication

Article • 01/11/2024

The System.Web adapters' remote authentication feature allows an ASP.NET Core app to determine a user's identity (authenticate an HTTP request) by deferring to an ASP.NET app. Enabling the feature adds an endpoint to the ASP.NET app that returns a serialized ClaimsPrincipal representing the authenticated user for any requests made to the endpoint. The ASP.NET Core app, then, registers a custom authentication handler that will (for endpoints with remote authentication enabled) determine a user's identity by calling that endpoint on the ASP.NET app and passing selected headers and cookies from the original request received by the ASP.NET Core app.

Configuration

There are just a few small code changes needed to enable remote authentication in a solution that's already set up according to the Getting Started.

First, follow the remote app setup instructions to connect the ASP.NET Core and ASP.NET apps. Then, there are just a couple extra extension methods to call to enable remote app authentication.

ASP.NET app configuration

The ASP.NET app needs to be configured to add the authentication endpoint. Adding the authentication endpoint is done by calling the `AddAuthenticationServer` extension method to set up the HTTP module that watches for requests to the authentication endpoint. Note that remote authentication scenarios typically want to add proxy support as well, so that any authentication related redirects correctly route to the ASP.NET Core app rather than the ASP.NET one.

C#

```csharp
SystemWebAdapterConfiguration.AddSystemWebAdapters(this)
    .AddProxySupport(options => options.UseForwardedHeaders = true)
    .AddRemoteAppServer(options =>
    {
        options.ApiKey = 
ConfigurationManager.AppSettings["RemoteAppApiKey"];
    })
    .AddAuthenticationServer();
```

ASP.NET Core app configuration

Next, the ASP.NET Core app needs to be configured to enable the authentication handler that will authenticate users by making an HTTP request to the ASP.NET app. Again, this is done by calling `AddAuthenticationClient` when registering System.Web adapters services:

```C#
builder.Services.AddSystemWebAdapters()
    .AddRemoteAppClient(options =>
    {
        options.RemoteAppUrl = new Uri(builder.Configuration

["ReverseProxy:Clusters:fallbackCluster:Destinations:fallbackApp:Address"]);
        options.ApiKey = builder.Configuration["RemoteAppApiKey"];
    })
    .AddAuthenticationClient(true);
```

The boolean that is passed to the `AddAuthenticationClient` call specifies whether remote app authentication should be the default authentication scheme. Passing `true` will cause the user to be authenticated via remote app authentication for all requests, whereas passing `false` means that the user will only be authenticated with remote app authentication if the remote app scheme is specifically requested (with `[Authorize(AuthenticationSchemes = RemoteAppAuthenticationDefaults.AuthenticationScheme)]` on a controller or action method, for example). Passing false for this parameter has the advantage of only making HTTP requests to the original ASP.NET app for authentication for endpoints that require remote app authentication but has the disadvantage of requiring annotating all such endpoints to indicate that they will use remote app auth.

In addition to the require boolean, an optional callback may be passed to `AddAuthenticationClient` to modify some other aspects of the remote authentication process's behavior:

- `RequestHeadersToForward`: This property contains headers that should be forwarded from a request when calling the authenticate API. By default, the only headers forwarded are `Authorization` and `Cookie`. Additional headers can be forwarded by adding them to this list. Alternatively, if the list is cleared (so that no headers are specified), then all headers will be forwarded.
- `ResponseHeadersToForward`: This property lists response headers that should be propagated back from the authenticate request to the original call that prompted

authentication in scenarios where identity is challenged. By default, this includes `Location`, `Set-Cookie`, and `WWW-Authenticate` headers.

- `AuthenticationEndpointPath`: The endpoint on the ASP.NET app where authenticate requests should be made. This defaults to `/systemweb-adapters/authenticate` and must match the endpoint specified in the ASP.NET authentication endpoint configuration.

Finally, if the ASP.NET Core app didn't previously include authentication middleware, that will need to be enabled (after routing middleware, but before authorization middleware):

C#

```
app.UseAuthentication();
```

Design

1. When requests are processed by the ASP.NET Core app, if remote app authentication is the default scheme or specified by the request's endpoint, the `RemoteAuthenticationAuthHandler` will attempt to authenticate the user.
   a. The handler will make an HTTP request to the ASP.NET app's authenticate endpoint. It will copy configured headers from the current request onto this new one in order to forward auth-relevant data. As mentioned above, default behavior is to copy the `Authorize` and `Cookie` headers. The API key header is also added for security purposes.
2. The ASP.NET app will serve requests sent to the authenticate endpoint. As long as the API keys match, the ASP.NET app will return either the current user's ClaimsPrincipal serialized into the response body or it will return an HTTP status code (like 401 or 302) and response headers indicating failure.
3. When the ASP.NET Core app's `RemoteAuthenticationAuthHandler` receives the response from the ASP.NET app:
   a. If a ClaimsPrincipal was successfully returned, the auth handler will deserialize it and use it as the current user's identity.
   b. If a ClaimsPrincipal was not successfully returned, the handler will store the result and if authentication is challenged (because the user is accessing a protected resource, for example), the request's response will be updated with the status code and selected response headers from the response from the authenticate endpoint. This enables challenge responses (like redirects to a login page) to be propagated to end users.

i. Because results from the ASP.NET app's authenticate endpoint may include data specific to that endpoint, users can register `IRemoteAuthenticationResultProcessor` implementations with the ASP.NET Core app which will run on any authentication results before they are used. As an example, the one built-in `IRemoteAuthenticationResultProcessor` is `RedirectUrlProcessor` which looks for `Location` response headers returned from the authenticate endpoint and ensures that they redirect back to the host of the ASP.NET Core app and not the ASP.NET app directly.

Known limitations

This remote authentication approach has a couple known limitations:

1. Because Windows authentication depends on a handle to a Windows identity, Windows authentication is not supported by this feature. Future work is planned to explore how shared Windows authentication might work. See dotnet/systemweb-adapters#246 for more information.
2. This feature allows the ASP.NET Core app to make use of an identity authenticated by the ASP.NET app, but all actions related to users (logging on, logging off, etc.) still need to be routed through the ASP.NET app.

Alternatives

If authentication in the ASP.NET app is done using `Microsoft.Owin` Cookie Authentication Middleware, an alternative solution to sharing identity is to configure the ASP.NET and ASP.NET Core apps so that they are able to share an authentication cookie. Sharing an authentication cookie enables:

- Both apps to determine the user identity from the same cookie.
- Signing in or out of one app signs the user in or out of the other app.

Note that because signing in typically depends on a specific database, not all authentication functionality will work in both apps:

- Users should sign in through only one of the apps, either the ASP.NET or ASP.NET Core app, whichever the database is setup to work with.
- Both apps are able to see the users' identity and claims.
- Both apps are able to sign the user out.

Details on how to configure sharing auth cookies between ASP.NET and ASP.NET Core apps are available in cookie sharing documentation. The following samples in the

System.Web adapters GitHub repo demonstrates remote app authentication with shared cookie configuration enabling both apps to sign users in and out :

- ASP.NET app
- ASP.NET Core app

Sharing authentication is a good option if both the following are true:

- The ASP.NET app is already using `Microsoft.Owin` cookie authentication.
- It's possible to update the ASP.NET app and ASP.NET Core apps to use matching data protection settings. Matching shared data protection settings includes a shared file path, Redis cache, or Azure Blob Storage for storing data protection keys.

For other scenarios, the remote authentication approach described previously in this doc is more flexible and is probably a better fit.

Unit Testing

Article • 01/11/2024

In most cases, there's no need to set up additional components for running tests. But if the component being tested uses HttpRuntime, it might be necessary to start up the `SystemWebAdapters` service, as shown in the following example:

C#

```csharp
namespace TestProject1;

/// <summary>
/// This demonstrates an xUnit feature that ensures all tests
/// in classes marked with this collection are run sequentially.
/// </summary>
[CollectionDefinition(nameof(SystemWebAdaptersHostedTests),
    DisableParallelization = true)]
public class SystemWebAdaptersHostedTests
{
}
[Collection(nameof(SystemWebAdaptersHostedTests))]
public class RuntimeTests
{
    /// <summary>
    /// This method starts up a host in the background that
    /// makes it possible to initialize <see cref="HttpRuntime"/>
    /// and <see cref="HostingEnvironment"/> with values needed
    /// for testing with the <paramref name="configure"/> option.
    /// </summary>
    /// <param name="configure">
    /// Configuration for the hosting and runtime options.
    /// </param>
    public static async Task<IDisposable> EnableRuntimeAsync(
        Action<SystemWebAdaptersOptions>? configure = null,
        CancellationToken token = default)
        => await new HostBuilder()
           .ConfigureWebHost(webBuilder =>
           {
               webBuilder
                   .UseTestServer()
                   .ConfigureServices(services =>
                   {
                       services.AddSystemWebAdapters();
                       if (configure is not null)
                       {
                           services.AddOptions
                               <SystemWebAdaptersOptions>()
                               .Configure(configure);
                       }
                   })
                   .Configure(app =>
                    {
                        // No need to configure pipeline for tests
                    });
            })
            .StartAsync(token);
    [Fact]
    public async Task RuntimeEnabled()
    {
        using (await EnableRuntimeAsync(options =>
            options.AppDomainAppPath = "path"))
        {
            Assert.True(HostingEnvironment.IsHosted);
            Assert.Equal("path", HttpRuntime.AppDomainAppPath);
        }
        Assert.False(HostingEnvironment.IsHosted);
    }
}
```

The tests must be executed in sequence, not in parallel. The preceding example illustrates how to achieve this by setting XUnit's DisableParallelization option to `true`. This setting disables parallel execution for a specific test collection, ensuring that the tests within that collection run one after the other, without concurrency.

A/B Testing endpoints during migration

Article • 01/11/2024

During incremental migration, new endpoints are brought over to a YARP enabled ASP.NET Core app. With the default setup, these endpoints are automatically served for all requests once deployed. In order to test these endpoints, or be able to turn them off if needed, additional setup is needed.

This document describes how to setup a conditional endpoint selection system to enable A/B testing during incremental migration. It assumes a setup as described in incremental migration overview as a starting point.

Conditional endpoint selection

To enable conditional endpoint selection, a few services need to be defined:

1. Metadata that can be added to an endpoint to turn on any conditional related logic. Endpoints include controllers, minimal APIs, Razor Page, etc. If this metadata isn't added to an endpoint, no conditional checks are performed for it.

   CSharp

   ```csharp
   [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
   public sealed class ConditionalRouteAttribute : Attribute
   {
       public bool IsConditional { get; set; } = true;
   }
   ```

   The property `IsConditional` is useful to set it `true` at a global level, and then allow lower levels, such as controllers or routes, to turn off the conditional checks.

2. The API we want to implement to make a decision per request and endpoint selection:

   CSharp

   ```csharp
   public interface IConditionalEndpointSelector
   {
       ValueTask<bool> IsEnabledAsync(HttpContext context, Endpoint
   candidate);
   }
   ```

3. A `MatcherPolicy` that is used to hook into routing and call the custom selector:

CSharp

```csharp
public static class ConditionalEndpointExtensions
{
    /// <summary>
    /// Registers a <see cref="IConditionalEndpointSelector"/> that is
called when selecting
    /// an endpoint that has been marked as conditional by <see
cref="WithConditionalRoute{TBuilder}(TBuilder)"/>.
    /// </summary>
    /// <typeparam name="T">Type of selector</typeparam>
    /// <param name="services">Service collection to add to.</param>
    public static void AddConditionalEndpoints<T>(this
IServiceCollection services)
        where T : class, IConditionalEndpointSelector
    {
        services.AddTransient<IConditionalEndpointSelector, T>();

services.TryAddEnumerable(ServiceDescriptor.Singleton<MatcherPolicy,
ConditionalEndpointMatcherPolicy>());
    }

    /// <summary>
    /// Enable conditional behavior for supplied endpoints. An
implementation of <see cref="IConditionalEndpointSelector"/> must be
registered as a service for this to be enabled at runtime.
    /// </summary>
    /// <param name="builder">The endpoint convention builder</param>
    /// <returns>The original convention builder.</returns>
    public static TBuilder WithConditionalRoute<TBuilder>(this TBuilder
builder)
        where TBuilder : IEndpointConventionBuilder
    {
        ArgumentNullException.ThrowIfNull(builder);

        return builder.WithMetadata(new ConditionalRouteAttribute());
    }

    private sealed class ConditionalEndpointMatcherPolicy :
MatcherPolicy, IEndpointSelectorPolicy
    {
        private readonly IConditionalEndpointSelector _selector;

        public
ConditionalEndpointMatcherPolicy(IConditionalEndpointSelector selector)
        {
            _selector = selector;
        }

        public override int Order => 0;

        public bool AppliesToEndpoints(IReadOnlyList<Endpoint>
endpoints)
            => endpoints.Any(e =>
e.Metadata.GetMetadata<ConditionalRouteAttribute>() is { IsConditional: 
true });

        public async Task ApplyAsync(HttpContext httpContext, 
CandidateSet candidates)
        {
            for (int i = 0; i < candidates.Count; i++)
            {
                var endpoint = candidates[i].Endpoint;

                if 
(endpoint.Metadata.GetMetadata<ConditionalRouteAttribute>() is { 
IsConditional: true })
                {
                    if (await _selector.IsEnabledAsync(httpContext, 
endpoint) == false)
                    {
                        candidates.SetValidity(i, false);
                    }
                }
            }
        }
    }
}
```

4. Implement a custom selector. As an example, this defines a check for the presence of a query parameter (`IgnoreLocal`) that turns off the local endpoint and uses the YARP endpoint instead.

CSharp

```csharp
public class QueryConditionalSelector : IConditionalEndpointSelector
{
    public ValueTask<bool> IsEnabledAsync(HttpContext context, Endpoint 
candidate)
    {
        var result = context.Request.Query.TryGetValue("IgnoreLocal", 
out var values) &&
            values is { Count: 1 } &&
            bool.TryParse(values[0], out var skip)
            && skip;

        return ValueTask.FromResult(result);
    }
```

5. Register the services in the program startup and mark the controllers for conditional selection. This marking can also be done using attribute marking on controllers or routes if needed.

diff

```
using Microsoft.AspNetCore.SystemWebAdapters;

var builder = WebApplication.CreateBuilder();
builder.Services.AddReverseProxy().LoadFromConfig(builder.Configuration
.GetSection("ReverseProxy"));

// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddSystemWebAdapters();
+ builder.Services.AddConditionalEndpoint<QueryConditionalSelector>();

var app = builder.Build();

app.UseSystemWebAdapters();

- app.MapDefaultControllerRoute();
+ app.MapDefaultControllerRoute().WithConditionalRoute();
app.MapReverseProxy();

app.Run();
```

Upgrade from ASP.NET MVC and Web API to ASP.NET Core MVC

Article • 07/18/2023

This article shows how to upgrade an ASP.NET Framework MVC or Web API app to ASP.NET Core MVC using the Visual Studio .NET Upgrade Assistant and the incremental update approach.

Upgrade using the .NET Upgrade Assistant

If your .NET Framework project has supporting libraries in the solution that are required, they should be upgraded to .NET Standard 2.0, if possible. For more information, see Upgrade supporting libraries.

1. Install the .NET Upgrade Assistant Visual Studio extension.
2. Open the ASP.NET MVC or Web API solution in Visual Studio.
3. In **Solution Explorer**, right click on the project to upgrade and select **Upgrade**. Select **Side-by-side incremental project upgrade**, which is the only upgrade option.
4. For the upgrade target, select **New project**.
5. Name the project and select the template. If the project you're migrating is a API project, select **ASP.NET Core Web API**. If it's an MVC project or MVC and Web API, select **ASP.NET Core MVC**.
6. Select **Next**
7. Select the target framework version and then select **Next**. For more information, see .NET and .NET Core Support Policy.
8. Review the **Summary of changes**, then select **Finish**.
9. The **Summary** step displays `<Framework Project>` **is now connected to** `<Framework ProjectCore>` **via Yarp proxy.** and a pie chart showing the migrated endpoints. Select **Upgrade Controller** and then select a controller to upgrade.
10. Select the component to upgrade, then select **Upgrade selection**.

Incremental update

Follow the steps in Get started with incremental ASP.NET to ASP.NET Core migration to continue the update process.

Upgrade an ASP.NET Framework Web Forms app to ASP.NET Core MVC

Article • 01/18/2024

This article shows how to upgrade an ASP.NET Framework Web Forms to ASP.NET Core MVC using the Visual Studio .NET Upgrade Assistant and the incremental update approach.

If your .NET Framework project has supporting libraries in its solution that are required, they should be upgraded to .NET Standard 2.0, if possible. For more information, see Upgrade supporting libraries.

1. Install the .NET Upgrade Assistant Visual Studio extension.
2. Open the ASP.NET Web Forms solution in Visual Studio.
3. In **Solution Explorer**, right click on the project to upgrade and select **Upgrade**. Select **Side-by-side incremental project upgrade**, which is the only upgrade option.
4. For the upgrade target, select **New project**.
5. Name the project and select the **ASP.NET Core** template and then select **Next**
6. Select the target framework version and then select **Next**. For more information, see .NET and .NET Core Support Policy.
7. Select **Done**, then select **Finish**.
8. The **Summary** step displays `<Framework Project>` **is now connected to** `<Framework ProjectCore>` **via Yarp proxy.**.
9. Select the component to upgrade, then select **Upgrade selection**.

Incremental update

Follow the steps in Get started with incremental ASP.NET to ASP.NET Core migration to continue the update process.

Migrate from ASP.NET Web API to ASP.NET Core

Article • 06/18/2024

ASP.NET Core combines ASP.NET 4.x's MVC and Web API app models into a single programming model known as ASP.NET Core MVC.

This article shows how to migrate the Products controller created in Getting Started with ASP.NET Web API 2 to ASP.NET Core.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.
- .NET 6.0 SDK

Create the new ASP.NET Core Web API project

Visual Studio

1. From the **File** menu, select **New** > **Project**.
2. Enter *Web API* in the search box.
3. Select the **ASP.NET Core Web API** template and select **Next**.
4. In the **Configure your new project** dialog, name the project *ProductsCore* and select **Next**.
5. In the **Additional information** dialog:
   a. Confirm the **Framework** is **.NET 6.0 (Long-term support)**.
   b. Confirm the checkbox for **Use controllers(uncheck to use minimal APIs)** is checked.
   c. Uncheck **Enable OpenAPI support**.
   d. Select **Create**.

Remove the *WeatherForecast* template files

1. Remove the `WeatherForecast.cs` and `Controllers/WeatherForecastController.cs` example files from the new *ProductsCore* project.
2. Open *Properties\launchSettings.json*.
3. Change `launchUrl` properties from `weatherforcast` to `productscore`.

The configuration for ASP.NET Core Web API

ASP.NET Core doesn't use the *App_Start* folder or the *Global.asax* file. The *web.config* file is added at publish time. For more information, see web.config file.

The `Program.cs` file:

- Replaces *Global.asax*.
- Handles all app startup tasks.

For more information, see App startup in ASP.NET Core.

The following shows the application startup code in the ASP.NET Core `Program.cs` file:

C#

```csharp
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Copy the *Product* model

Visual Studio

1. In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.

2. Right-click the **Models** folder. Select **Add** > **Class**. Name the class *Product* and select **Add**.
3. Replace the template model code with the following:

C#

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

The preceding highlighted code changes the following:

- The `?` annotation has been added to declare the `Name` and `Category` properties as nullable reference types.

By utilizing the Nullable feature introduced in C# 8, ASP.NET Core can provide additional code flow analysis and compile-time safety in the handling of reference types. For example, protecting against `null` reference exceptions.

In this case, the intent is that the `Name` and `Category` can be nullable types.

ASP.NET Core 6.0 projects enable nullable reference types by default. For more information, see Nullable reference types.

Copy the *ProductsController*

Visual Studio

1. Right-click the **Controllers** folder.
2. Select **Add** > **Controller...**.
3. In **Add New Scaffolded Item** dialog, select **Mvc Controller - Empty** then select **Add**.
4. Name the controller *ProductsController* and select **Add**.
5. Replace the template controller code with the following:

C#

```csharp
using Microsoft.AspNetCore.Mvc;
using ProductsCore.Models;

namespace ProductsCore.Controllers;

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

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

    [HttpGet("{id}")]
    public ActionResult<Product> GetProduct(int id)
    {
        var product = products.FirstOrDefault((p) => p.Id == id);
        if (product == null)
        {
            return NotFound();
        }
        return product;
    }
}
```

The preceding highlighted code changes the following, to migrate to ASP.NET Core:

- Removes using statements for the following ASP.NET 4.x components that don't exist in ASP.NET Core:
  - `ApiController` class

    - `System.Web.Http` namespace
    - `IHttpActionResult` interface

- Changes the `using ProductsApp.Models;` statement to `using ProductsCore.Models;`.

- Sets the root namespace to `ProductsCore`.

- Changes `ApiController` to ControllerBase.

- Adds `using Microsoft.AspNetCore.Mvc;` to resolve the `ControllerBase` reference.

- Changes the `GetProduct` action's return type from `IHttpActionResult` to `ActionResult<Product>`. For more info, see Controller action return types.

- Simplifies the `GetProduct` action's `return` statement to the following statement:

  C#

  ```csharp
  return product;
  ```

- Adds the following attributes which are explained in the next sections:
    - `[Route("api/[controller]")]`
    - `[ApiController]`
    - `[HttpGet]`
    - `[HttpGet("{id}")]`

Routing

ASP.NET Core provides a minimal hosting model in which the endpoint routing middleware wraps the entire middleware pipeline, therefore routes can be added directly to the WebApplication without an explicit call to UseEndpoints or UseRouting to register routes.

`UseRouting` can still be used to specify where route matching happens, but `UseRouting` doesn't need to be explicitly called if routes should be matched at the beginning of the middleware pipeline.

C#

```csharp
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
```

```
builder.Services.AddControllers();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Note: Routes added directly to the WebApplication execute at the ***end*** of the pipeline.

Routing in the migrated `ProductsController`

The migrated `ProductsController` contains the following highlighted attributes:

```C#
using Microsoft.AspNetCore.Mvc;
using ProductsCore.Models;

namespace ProductsCore.Controllers;

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

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

    [HttpGet("{id}")]
    public ActionResult<Product> GetProduct(int id)
    {
        var product = products.FirstOrDefault((p) => p.Id == id);
        if (product == null)
        {
            return NotFound();
        }
        return product;
    }
}
```

- The [Route] attribute configures the controller's attribute routing pattern.
- The [ApiController] attribute makes attribute routing a requirement for all actions in this controller.
- Attribute routing supports tokens, such as `[controller]` and `[action]`. At runtime, each token is replaced with the name of the controller or action, respectively, to which the attribute has been applied. The tokens:
  - Reduces or eliminates the need to use hard coded strings for the route.
  - Ensure routes remain synchronized with the corresponding controllers and actions when automatic rename refactorings are applied.
- HTTP Get requests are enabled for `ProductController` actions with the following attributes:
  - [HttpGet] attribute applied to the `GetAllProducts` action.
  - `[HttpGet("{id}")]` attribute applied to the `GetProduct` action.

Run the migrated project, and browse to `/api/products`. For example: https://localhost:`<port>`/api/products. A full list of three products appears. Browse to `/api/products/1`. The first product appears.

View or download sample code (how to download)

Additional resources

- Create web APIs with ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Compatibility version for ASP.NET Core MVC

Migrate configuration to ASP.NET Core

Article • 10/04/2024

In the previous article, we began to migrate an ASP.NET MVC project to ASP.NET Core MVC. In this article, we migrate configuration.

View or download sample code (how to download)

Setup configuration

ASP.NET Core no longer uses the *Global.asax* and *web.config* files that previous versions of ASP.NET utilized. In the earlier versions of ASP.NET, application startup logic was placed in an `Application_StartUp` method within *Global.asax*. Later, in ASP.NET MVC, a `Startup.cs` file was included in the root of the project; and, it was called when the application started. ASP.NET Core has adopted this approach completely by placing all startup logic in the `Startup.cs` file.

The *web.config* file has also been replaced in ASP.NET Core. Configuration itself can now be configured, as part of the application startup procedure described in `Startup.cs`. Configuration can still utilize XML files, but typically ASP.NET Core projects will place configuration values in a JSON-formatted file, such as `appsettings.json`. ASP.NET Core's configuration system can also easily access environment variables, which can provide a more secure and robust location for environment-specific values. This is especially true for secrets like connection strings and API keys that shouldn't be checked into source control. See Configuration to learn more about configuration in ASP.NET Core.

⚠ **Warning**

This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

For this article, we are starting with the partially migrated ASP.NET Core project from the previous article. To setup configuration, add the following constructor and property to the `Startup.cs` file located in the root of the project:

C#

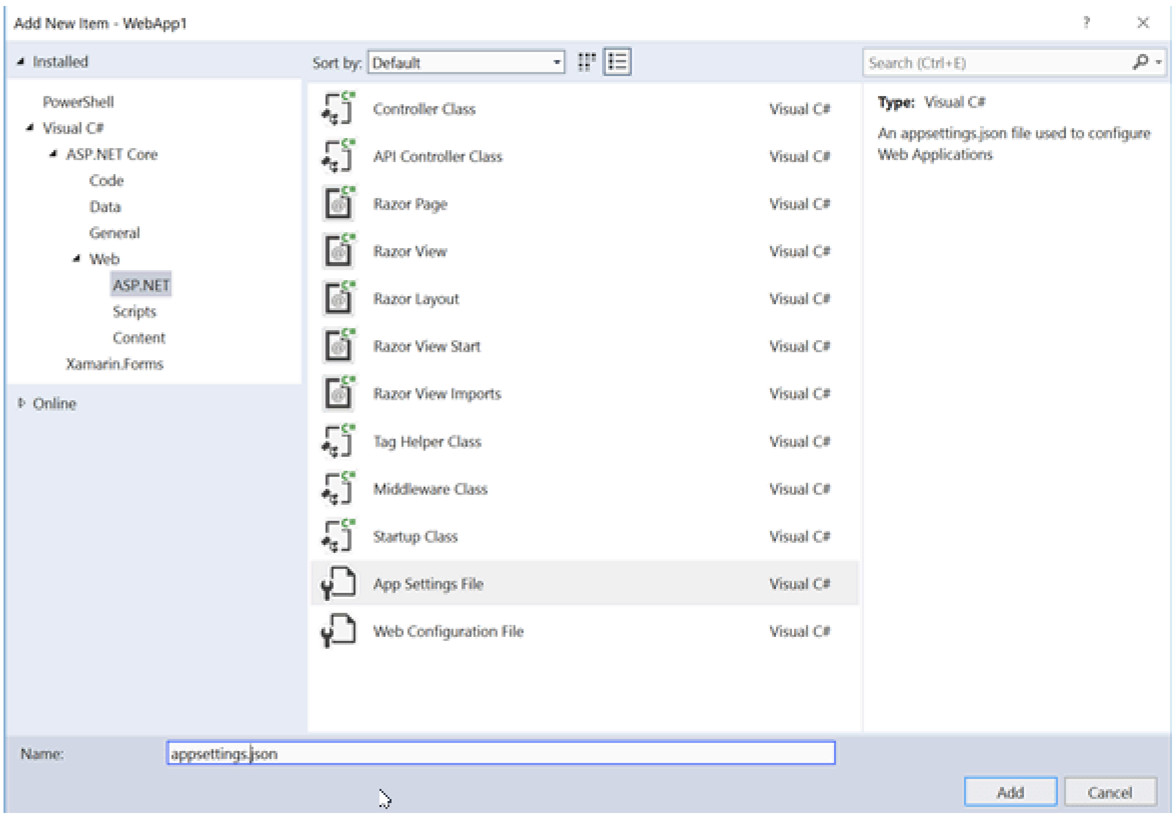
```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

Note that at this point, the `Startup.cs` file won't compile, as we still need to add the following `using` statement:

C#

```
using Microsoft.Extensions.Configuration;
```

Add an `appsettings.json` file to the root of the project using the appropriate item template:


Add New Item - WebApp1

Installed
- PowerShell
- Visual C#
  - ASP.NET Core
    - Code
    - Data
    - General
    - Web
      - ASP.NET
      - Scripts
      - Content
  - Xamarin.Forms
- Online

Sort by: Default

Controller Class	Visual C#
API Controller Class	Visual C#
Razor Page	Visual C#
Razor View	Visual C#
Razor Layout	Visual C#
Razor View Start	Visual C#
Razor View Imports	Visual C#
Tag Helper Class	Visual C#
Middleware Class	Visual C#
Startup Class	Visual C#
App Settings File	Visual C#
Web Configuration File	Visual C#

Type: Visual C#

An appsettings.json file used to configure Web Applications

Name: appsettings.json

Migrate configuration settings from web.config

Our ASP.NET MVC project included the required database connection string in *web.config*, in the `<connectionStrings>` element. In our ASP.NET Core project, we are going to store this information in the `appsettings.json` file. Open `appsettings.json`, and note that it already includes the following:

JSON

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

In the highlighted line depicted above, change the name of the database from **_CHANGE_ME** to the name of your database.

Summary

ASP.NET Core places all startup logic for the application in a single file, in which the necessary services and dependencies can be defined and configured. It replaces the *web.config* file with a flexible configuration feature that can leverage a variety of file formats, such as JSON, as well as environment variables.

Migrate Authentication and Identity to ASP.NET Core

Article • 10/30/2024

By Steve Smith

In the previous article, we migrated configuration from an ASP.NET MVC project to ASP.NET Core MVC. In this article, we migrate the registration, login, and user management features.

Configure Identity and Membership

In ASP.NET MVC, authentication and identity features are configured using ASP.NET Identity in `Startup.Auth.cs` and `IdentityConfig.cs`, located in the *App_Start* folder. In ASP.NET Core MVC, these features are configured in `Startup.cs`.

Install the following NuGet packages:

- `Microsoft.AspNetCore.Identity.EntityFrameworkCore`
- `Microsoft.AspNetCore.Authentication.Cookies`
- `Microsoft.EntityFrameworkCore.SqlServer`

⚠ **Warning**

This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

In `Startup.cs`, update the `Startup.ConfigureServices` method to use Entity Framework and Identity services:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    // Add EF services to the services container.
    services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))
);

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

     services.AddMvc();
}
```

At this point, there are two types referenced in the above code that we haven't yet migrated from the ASP.NET MVC project: `ApplicationDbContext` and `ApplicationUser`. Create a new *Models* folder in the ASP.NET Core project, and add two classes to it corresponding to these types. You will find the ASP.NET MVC versions of these classes in `/Models/IdentityModels.cs`, but we will use one file per class in the migrated project since that's more clear.

`ApplicationUser.cs`:

```C#
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

namespace NewMvcProject.Models
{
    public class ApplicationUser : IdentityUser
    {
    }
}
```

`ApplicationDbContext.cs`:

```C#
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.Data.Entity;

namespace NewMvcProject.Models
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
            // Customize the ASP.NET Core Identity model and override the
defaults if needed.
            // For example, you can rename the ASP.NET Core Identity table
names and more.
            // Add your customizations after calling
base.OnModelCreating(builder);
        }
    }
}
```

The ASP.NET Core MVC Starter Web project doesn't include much customization of users, or the `ApplicationDbContext`. When migrating a real app, you also need to migrate all of the custom properties and methods of your app's user and `DbContext` classes, as well as any other Model classes your app utilizes. For example, if your `DbContext` has a `DbSet<Album>`, you need to migrate the `Album` class.

With these files in place, the `Startup.cs` file can be made to compile by updating its `using` statements:

C#

```csharp
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
```

Our app is now ready to support authentication and Identity services. It just needs to have these features exposed to users.

Migrate registration and login logic

With Identity services configured for the app and data access configured using Entity Framework and SQL Server, we're ready to add support for registration and login to the app. Recall that earlier in the migration process we commented out a reference to *_LoginPartial* in `_Layout.cshtml`. Now it's time to return to that code, uncomment it, and add in the necessary controllers and views to support login functionality.

Uncomment the `@Html.Partial` line in `_Layout.cshtml`:

CSHTML

```
        <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
      </ul>
      @*@Html.Partial("_LoginPartial")*@
    </div>
  </div>
```

Now, add a new Razor view called *_LoginPartial* to the *Views/Shared* folder:

Update `_LoginPartial.cshtml` with the following code (replace all of its contents):

CSHTML

```
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="" asp-controller="Account" asp-action="Logout"
method="post" id="logoutForm" class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="" asp-controller="Manage" asp-action="Index"
title="Manage">Hello @UserManager.GetUserName(User)!</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn navbar-
link">Log out</button>
            </li>
        </ul>
    </form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="" asp-controller="Account" asp-
action="Register">Register</a></li>
        <li><a asp-area="" asp-controller="Account" asp-action="Login">Log
in</a></li>
    </ul>
}
```

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

Summary

ASP.NET Core introduces changes to the ASP.NET Identity features. In this article, you have seen how to migrate the authentication and user management features of ASP.NET Identity to ASP.NET Core.

Migrate from ClaimsPrincipal.Current

Article • 06/18/2024

In ASP.NET 4.x projects, it was common to use ClaimsPrincipal.Current to retrieve the current authenticated user's identity and claims. In ASP.NET Core, this property is no longer set. Code that was depending on it needs to be updated to get the current authenticated user's identity through a different means.

Context-specific state instead of static state

When using ASP.NET Core, the values of both `ClaimsPrincipal.Current` and `Thread.CurrentPrincipal` aren't set. These properties both represent static state, which ASP.NET Core generally avoids. Instead, ASP.NET Core uses dependency injection (DI) to provide dependencies such as the current user's identity. Getting the current user's identity from DI is more testable, too, since test identities can be easily injected.

Retrieve the current user in an ASP.NET Core app

There are several options for retrieving the current authenticated user's `ClaimsPrincipal` in ASP.NET Core in place of `ClaimsPrincipal.Current`:

- **ControllerBase.User**. MVC controllers can access the current authenticated user with their User property.
- **HttpContext.User**. Components with access to the current `HttpContext` (middleware, for example) can get the current user's `ClaimsPrincipal` from HttpContext.User.
- **Passed in from caller**. Libraries without access to the current `HttpContext` are often called from controllers or middleware components and can have the current user's identity passed as an argument.
- **IHttpContextAccessor**. The project being migrated to ASP.NET Core may be too large to easily pass the current user's identity to all necessary locations. In such cases, IHttpContextAccessor can be used as a workaround. `IHttpContextAccessor` is able to access the current `HttpContext` (if one exists). If DI is being used, see Access HttpContext in ASP.NET Core. A short-term solution to getting the current

user's identity in code that hasn't yet been updated to work with ASP.NET Core's DI-driven architecture would be:

  - Make `IHttpContextAccessor` available in the DI container by calling AddHttpContextAccessor in `Startup.ConfigureServices`.
  - Get an instance of `IHttpContextAccessor` during startup and store it in a static variable. The instance is made available to code that was previously retrieving the current user from a static property.
  - Retrieve the current user's `ClaimsPrincipal` using `HttpContextAccessor.HttpContext?.User`. If this code is used outside of the context of an HTTP request, the `HttpContext` is null.

The final option, using an `IHttpContextAccessor` instance stored in a static variable, is contrary to the ASP.NET Core principle of preferring injected dependencies to static dependencies. Plan to eventually retrieve `IHttpContextAccessor` instances from DI instead. A static helper can be a useful bridge, though, when migrating large existing ASP.NET apps that use `ClaimsPrincipal.Current`.

Migrate from ASP.NET Membership authentication to ASP.NET Core 2.0 Identity

Article • 10/30/2024

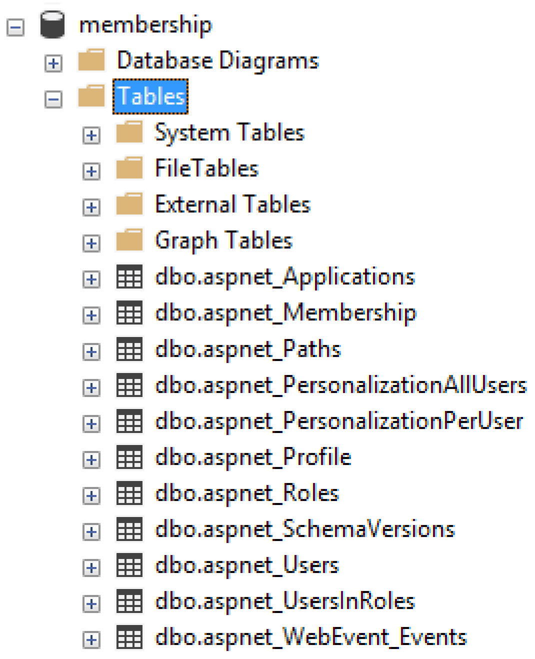
By Isaac Levin

This article demonstrates migrating the database schema for ASP.NET apps using Membership authentication to ASP.NET Core 2.0 Identity.

> **Note**
>
> This document provides the steps needed to migrate the database schema for ASP.NET Membership-based apps to the database schema used for ASP.NET Core Identity. For more information about migrating from ASP.NET Membership-based authentication to ASP.NET Identity, see Migrate an existing app from SQL Membership to ASP.NET Identity. For more information about ASP.NET Core Identity, see Introduction to Identity on ASP.NET Core.

Review of Membership schema

Prior to ASP.NET 2.0, developers were tasked with creating the entire authentication and authorization process for their apps. With ASP.NET 2.0, Membership was introduced, providing a boilerplate solution to handling security within ASP.NET apps. Developers were now able to bootstrap a schema into a SQL Server database with the ASP.NET SQL Server Registration Tool (`Aspnet_regsql.exe`) (no longer supported). After running this command, the following tables were created in the database.


membership
Database Diagrams
Tables
System Tables
FileTables
External Tables
Graph Tables
dbo.aspnet_Applications
dbo.aspnet_Membership
dbo.aspnet_Paths
dbo.aspnet_PersonalizationAllUsers
dbo.aspnet_PersonalizationPerUser
dbo.aspnet_Profile
dbo.aspnet_Roles
dbo.aspnet_SchemaVersions
dbo.aspnet_Users
dbo.aspnet_UsersInRoles
dbo.aspnet_WebEvent_Events

To migrate existing apps to ASP.NET Core 2.0 Identity, the data in these tables needs to be migrated to the tables used by the new Identity schema.

ASP.NET Core Identity 2.0 schema

ASP.NET Core 2.0 follows the Identity principle introduced in ASP.NET 4.5. Though the principle is shared, the implementation between the frameworks is different, even between versions of ASP.NET Core (see Migrate authentication and Identity to ASP.NET Core 2.0).

The fastest way to view the schema for ASP.NET Core 2.0 Identity is to create a new ASP.NET Core 2.0 app. Follow these steps in Visual Studio 2017:

1. Select **File** > **New** > **Project**.
2. Create a new **ASP.NET Core Web Application** project named *CoreIdentitySample*.
3. Select **ASP.NET Core 2.0** in the dropdown and then select **Web Application**. This template produces a Razor Pages app. Before clicking **OK**, click **Change Authentication**.
4. Choose **Individual User Accounts** for the Identity templates. Finally, click **OK**, then **OK**. Visual Studio creates a project using the ASP.NET Core Identity template.
5. Select **Tools** > **NuGet Package Manager** > **Package Manager Console** to open the **Package Manager Console** (PMC) window.
6. Navigate to the project root in PMC, and run the Entity Framework (EF) Core `Update-Database` command.

ASP.NET Core 2.0 Identity uses EF Core to interact with the database storing the authentication data. In order for the newly created app to work, there needs to be a database to store this data. After creating a new app, the fastest way to inspect the schema in a database environment is to create the database using EF Core Migrations. This process creates a database, either locally or elsewhere, which mimics that schema. Review the preceding documentation for more information.

EF Core commands use the connection string for the database specified in `appsettings.json`. The following connection string targets a database on *localhost* named *asp-net-core-identity*. In this setting, EF Core is configured to use the `DefaultConnection` connection string.

JSON

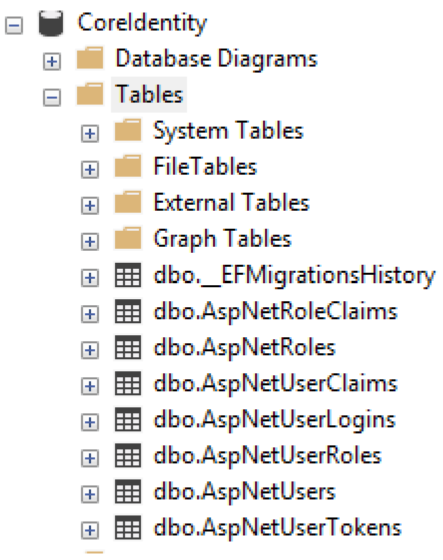
```json
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=aspnet-core-identity;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

⚠ **Warning**

This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

1. Select **View** > **SQL Server Object Explorer**. Expand the node corresponding to the database name specified in the `ConnectionStrings:DefaultConnection` property of `appsettings.json`.

   The `Update-Database` command created the database specified with the schema and any data needed for app initialization. The following image depicts the table structure that's created with the preceding steps.



Migrate the schema

There are subtle differences in the table structures and fields for both Membership and ASP.NET Core Identity. The pattern has changed substantially for authentication/authorization with ASP.NET and ASP.NET Core apps. The key objects that are still used with Identity are *Users* and *Roles*. Here are mapping tables for *Users*, *Roles*, and *UserRoles*.

Users

Expand table

Identity (`dbo.AspNetUsers`) column	**Type**	**Membership (`dbo.aspnet_Users` / `dbo.aspnet_Membership`) column**	**Type**
`Id`	`string`	`aspnet_Users.UserId`	`string`
`UserName`	`string`	`aspnet_Users.UserName`	`string`
`Email`	`string`	`aspnet_Membership.Email`	`string`
`NormalizedUserName`	`string`	`aspnet_Users.LoweredUserName`	`string`
`NormalizedEmail`	`string`	`aspnet_Membership.LoweredEmail`	`string`
`PhoneNumber`	`string`	`aspnet_Users.MobileAlias`	`string`
`LockoutEnabled`	`bit`	`aspnet_Membership.IsLockedOut`	`bit`

`IsLockedOut` doesn't map to `LockoutEnabled`. `IsLockedOut` is set if a user had too many failed logins and is locked out for a set time. `LockoutEnabled` enables locking out a user with too many failed sign in attempts. When the user has too many failed sign in attempts, `LockoutEnd` is set to a date in the future, and the user can't sign in until that date. If `LockoutEnabled` is false then a user is never locked out for too many failed sign in attempts. Per OWASP, *Temporary account lockout after several failed attempts is too simple of a target for DoS attacks against legitimate users*.

For more information on lock out, see OWASP Testing for Weak Lock Out Mechanism.

Apps migrating to Identity that wish to enable failed login lockout should set `LockoutEnabled` to true as part of the migration.

Note

Not all the field mappings resemble one-to-one relationships from Membership to ASP.NET Core Identity. The preceding table takes the default Membership User schema and maps it to the ASP.NET Core Identity schema. Any other custom fields that were used for Membership need to be mapped manually. In this mapping, there's no map for passwords, as both password criteria and password salts don't migrate between the two. **It's recommended to leave the password as null and to ask users to reset their passwords.** In ASP.NET Core Identity, `LockoutEnd` should be set to some date in the future if the user is locked out. This is shown in the migration script.

Roles

Expand table

Identity (`dbo.AspNetRoles`) column	Type	Membership (`dbo.aspnet_Roles`) column	Type
`Id`	`string`	`RoleId`	`string`
`Name`	`string`	`RoleName`	`string`
`NormalizedName`	`string`	`LoweredRoleName`	`string`

User Roles

Expand table

Identity (`dbo.AspNetUserRoles`) column	Type	Membership (`dbo.aspnet_UsersInRoles`) column	Type
`RoleId`	`string`	`RoleId`	`string`
`UserId`	`string`	`UserId`	`string`

Reference the preceding mapping tables when creating a migration script for *Users* and *Roles*. The following example assumes you have two databases on a database server. One database contains the existing ASP.NET Membership schema and data. The other *CoreIdentitySample* database was created using steps described earlier. Comments are included inline for more details.

SQL

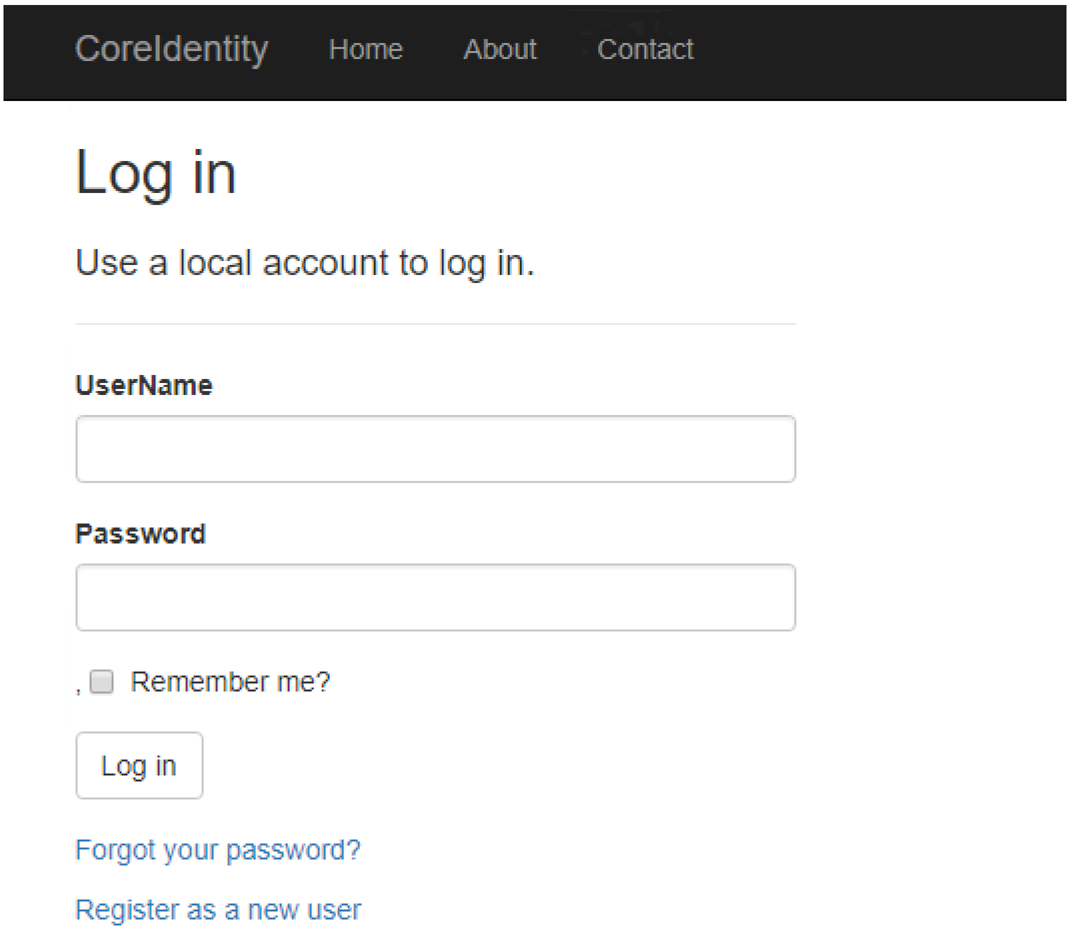
```sql
-- THIS SCRIPT NEEDS TO RUN FROM THE CONTEXT OF THE MEMBERSHIP DB
BEGIN TRANSACTION MigrateUsersAndRoles
USE aspnetdb

-- INSERT USERS
INSERT INTO CoreIdentitySample.dbo.AspNetUsers
            (Id,
             UserName,
             NormalizedUserName,
             PasswordHash,
             SecurityStamp,
             EmailConfirmed,
             PhoneNumber,
             PhoneNumberConfirmed,
             TwoFactorEnabled,
             LockoutEnd,
             LockoutEnabled,
             AccessFailedCount,
             Email,
             NormalizedEmail)
SELECT aspnet_Users.UserId,
       aspnet_Users.UserName,
       -- The NormalizedUserName value is upper case in ASP.NET Core
Identity
       UPPER(aspnet_Users.UserName),
       -- Creates an empty password since passwords don't map between the 2
schemas
       '',
       /*
        The SecurityStamp token is used to verify the state of an account
and
        is subject to change at any time. It should be initialized as a new
ID.
       */
       NewID(),
       /*
        EmailConfirmed is set when a new user is created and confirmed via
```

```
email.
        Users must have this set during migration to reset passwords.
       */
       1,
       aspnet_Users.MobileAlias,
       CASE
         WHEN aspnet_Users.MobileAlias IS NULL THEN 0
         ELSE 1
       END,
       -- 2FA likely wasn't setup in Membership for users, so setting as
false.
       0,
       CASE
         -- Setting lockout date to time in the future (1,000 years)
         WHEN aspnet_Membership.IsLockedOut = 1 THEN Dateadd(year, 1000,
                                                     Sysutcdatetime())
         ELSE NULL
       END,
       aspnet_Membership.IsLockedOut,
       /*
        AccessFailedAccount is used to track failed logins. This is stored
in
        Membership in multiple columns. Setting to 0 arbitrarily.
       */
       0,
       aspnet_Membership.Email,
       -- The NormalizedEmail value is upper case in ASP.NET Core Identity
       UPPER(aspnet_Membership.Email)
FROM   aspnet_Users
       LEFT OUTER JOIN aspnet_Membership
                    ON aspnet_Membership.ApplicationId =
                       aspnet_Users.ApplicationId
                       AND aspnet_Users.UserId = aspnet_Membership.UserId
       LEFT OUTER JOIN CoreIdentitySample.dbo.AspNetUsers
                    ON aspnet_Membership.UserId = AspNetUsers.Id
WHERE  AspNetUsers.Id IS NULL

-- INSERT ROLES
INSERT INTO CoreIdentitySample.dbo.AspNetRoles(Id, Name)
SELECT RoleId, RoleName
FROM aspnet_Roles;

-- INSERT USER ROLES
INSERT INTO CoreIdentitySample.dbo.AspNetUserRoles(UserId, RoleId)
SELECT UserId, RoleId
FROM aspnet_UsersInRoles;

IF @@ERROR <> 0
  BEGIN
    ROLLBACK TRANSACTION MigrateUsersAndRoles
    RETURN
  END

COMMIT TRANSACTION MigrateUsersAndRoles
```

After completion of the preceding script, the ASP.NET Core Identity app created earlier is populated with Membership users. Users need to change their passwords before logging in.

> ⓘ **Note**
>
> If the Membership system had users with user names that didn't match their email address, changes are required to the app created earlier to accommodate this. The default template expects `UserName` and `Email` to be the same. For situations in which they're different, the login process needs to be modified to use `UserName` instead of `Email`.

In the `PageModel` of the Login Page, located at , remove the `[EmailAddress]` attribute from the *Email* property. Rename it to *UserName*. This requires a change wherever `EmailAddress` is mentioned, in the *View* and *PageModel*. The result looks like the following:



Next steps

In this tutorial, you learned how to port users from SQL membership to ASP.NET Core 2.0 Identity. For more information regarding ASP.NET Core Identity, see Introduction to Identity.

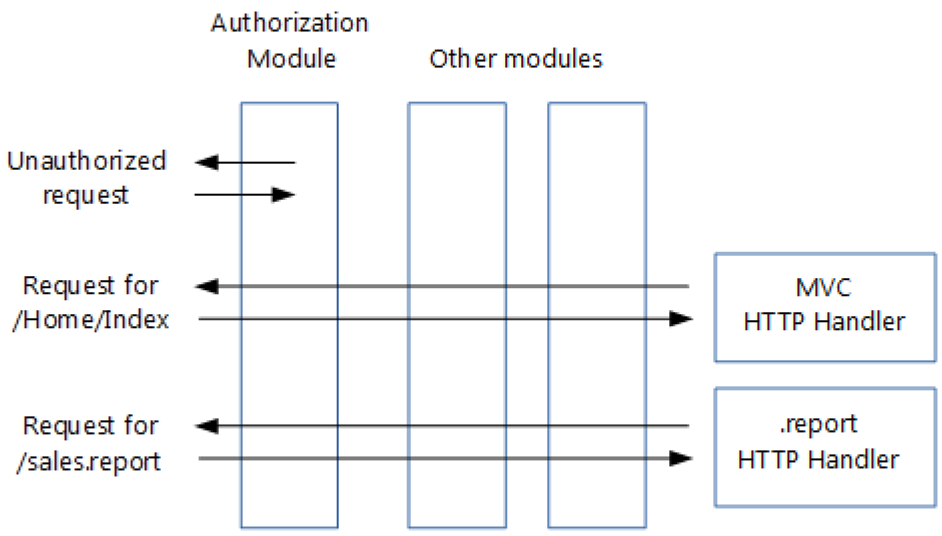
Migrate HTTP handlers and modules to ASP.NET Core middleware

Article • 03/23/2024

This article shows how to migrate existing ASP.NET HTTP modules and handlers from system.webserver to ASP.NET Core middleware.

Modules and handlers revisited

Before proceeding to ASP.NET Core middleware, let's first recap how HTTP modules and handlers work:



Handlers are:

- Classes that implement IHttpHandler
- Used to handle requests with a given file name or extension, such as *.report*
- Configured in *Web.config*

Modules are:

- Classes that implement IHttpModule
- Invoked for every request
- Able to short-circuit (stop further processing of a request)
- Able to add to the HTTP response, or create their own

- Configured in *Web.config*

The order in which modules process incoming requests is determined by:

1. A series events fired by ASP.NET, such as BeginRequest and AuthenticateRequest. For a complete list, see System.Web.HttpApplication. Each module can create a handler for one or more events.

2. For the same event, the order in which they're configured in *Web.config*.

In addition to modules, you can add handlers for the life cycle events to your `Global.asax.cs` file. These handlers run after the handlers in the configured modules.

From handlers and modules to middleware

Middleware are simpler than HTTP modules and handlers:

- Modules, handlers, `Global.asax.cs`, *Web.config* (except for IIS configuration) and the application life cycle are gone

- The roles of both modules and handlers have been taken over by middleware

- Middleware are configured using code rather than in *Web.config*

- Pipeline branching lets you send requests to specific middleware, based on not only the URL but also on request headers, query strings, etc.

Middleware are very similar to modules:

- Invoked in principle for every request

- Able to short-circuit a request, by not passing the request to the next middleware

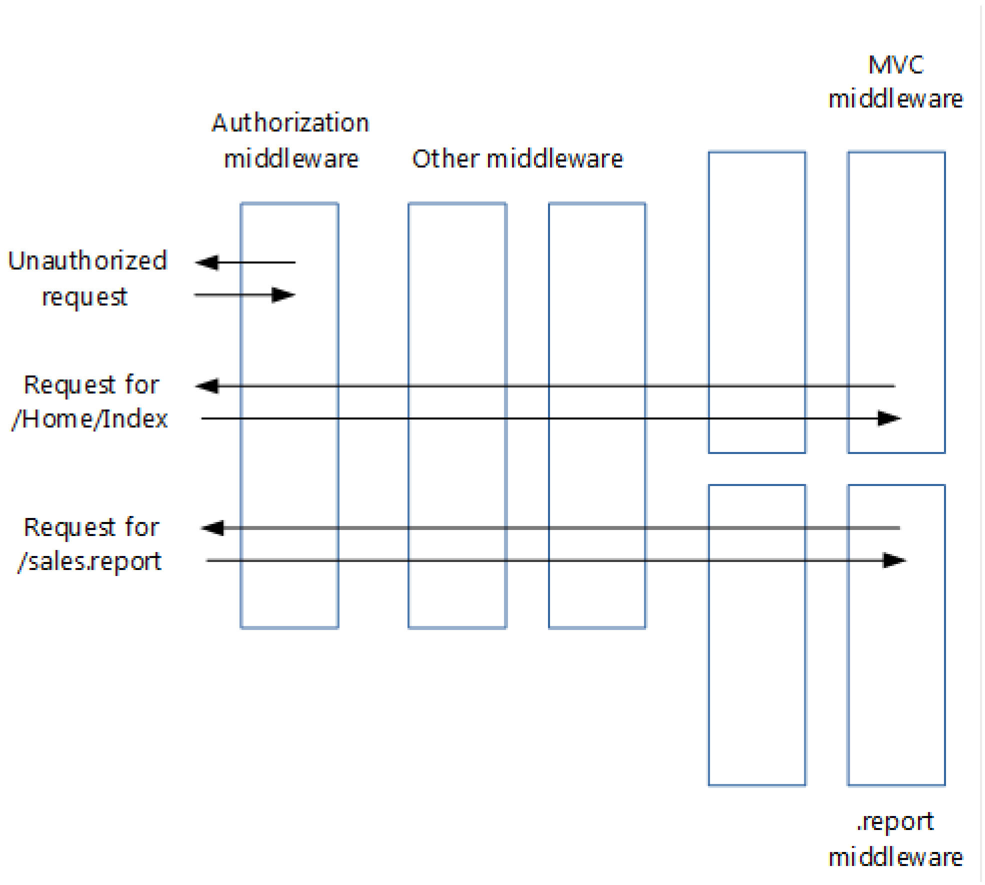
- Able to create their own HTTP response

Middleware and modules are processed in a different order:

- Order of middleware is based on the order in which they're inserted into the request pipeline, while order of modules is mainly based on System.Web.HttpApplication events.

- Order of middleware for responses is the reverse from that for requests, while order of modules is the same for requests and responses

- See Create a middleware pipeline with IApplicationBuilder



Note how in the image above, the authentication middleware short-circuited the request.

Migrating module code to middleware

An existing HTTP module will look similar to this:

```C#
// ASP.NET 4 module

using System;
using System.Web;

namespace MyApp.Modules
{
    public class MyModule : IHttpModule
    {
        public void Dispose()
        {
        }

        public void Init(HttpApplication application)
        {
            application.BeginRequest += (new
EventHandler(this.Application_BeginRequest));
            application.EndRequest += (new
EventHandler(this.Application_EndRequest));
```

```
        }

        private void Application_BeginRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the beginning of request
processing.
        }

        private void Application_EndRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the end of request processing.
        }
    }
}
```

As shown in the Middleware page, an ASP.NET Core middleware is a class that exposes an `Invoke` method taking an `HttpContext` and returning a `Task`. Your new middleware will look like this:

```C#
// ASP.NET Core middleware

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyMiddleware
    {
        private readonly RequestDelegate _next;

        public MyMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task Invoke(HttpContext context)
        {
            // Do something with context near the beginning of request
processing.

            await _next.Invoke(context);

            // Clean up.
        }
    }
```

```csharp
    public static class MyMiddlewareExtensions
    {
        public static IApplicationBuilder UseMyMiddleware(this
IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyMiddleware>();
        }
    }
}
```

The preceding middleware template was taken from the section on writing middleware.

The *MyMiddlewareExtensions* helper class makes it easier to configure your middleware in your `Startup` class. The `UseMyMiddleware` method adds your middleware class to the request pipeline. Services required by the middleware get injected in the middleware's constructor.

Your module might terminate a request, for example if the user isn't authorized:

C#

```csharp
// ASP.NET 4 module that may terminate the request

private void Application_BeginRequest(Object source, EventArgs e)
{
    HttpContext context = ((HttpApplication)source).Context;

    // Do something with context near the beginning of request processing.

    if (TerminateRequest())
    {
        context.Response.End();
        return;
    }
}
```

A middleware handles this by not calling `Invoke` on the next middleware in the pipeline. Keep in mind that this doesn't fully terminate the request, because previous middlewares will still be invoked when the response makes its way back through the pipeline.

C#

```csharp
// ASP.NET Core middleware that may terminate the request

public async Task Invoke(HttpContext context)
{
    // Do something with context near the beginning of request processing.
    if (!TerminateRequest())
        await _next.Invoke(context);

    // Clean up.
}
```

When you migrate your module's functionality to your new middleware, you may find that your code doesn't compile because the `HttpContext` class has significantly changed in ASP.NET Core. Later on, you'll see how to migrate to the new ASP.NET Core HttpContext.

Migrating module insertion into the request pipeline

HTTP modules are typically added to the request pipeline using *Web.config*:

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <modules>
      <add name="MyModule" type="MyApp.Modules.MyModule"/>
    </modules>
  </system.webServer>
</configuration>
```

Convert this by adding your new middleware to the request pipeline in your `Startup` class:

C#

```csharp
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

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

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions =
Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptio
ns>();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOpti
ons>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

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

The exact spot in the pipeline where you insert your new middleware depends on the event that it handled as a module (`BeginRequest`, `EndRequest`, etc.) and its order in your list of modules in *Web.config*.

As previously stated, there's no application life cycle in ASP.NET Core and the order in which responses are processed by middleware differs from the order used by modules. This could make your ordering decision more challenging.

If ordering becomes a problem, you could split your module into multiple middleware components that can be ordered independently.

Migrating handler code to middleware

An HTTP handler looks something like this:

C#

```
// ASP.NET 4 handler

using System.Web;

namespace MyApp.HttpHandlers
{
    public class MyHandler : IHttpHandler
    {
        public bool IsReusable { get { return true; } }

        public void ProcessRequest(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            context.Response.Output.Write(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.QueryString["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }
}
```

In your ASP.NET Core project, you would translate this to a middleware similar to this:

C#

```
// ASP.NET Core middleware migrated from a handler

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyHandlerMiddleware
    {

        // Must have constructor with this signature, otherwise exception at
run time
        public MyHandlerMiddleware(RequestDelegate next)
        {
            // This is an HTTP Handler, so no need to store next
        }

        public async Task Invoke(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            await context.Response.WriteAsync(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.Query["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }

    public static class MyHandlerExtensions
    {
        public static IApplicationBuilder UseMyHandler(this
IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyHandlerMiddleware>();
        }
    }
}
```

This middleware is very similar to the middleware corresponding to modules. The only real difference is that here there's no call to `_next.Invoke(context)`. That makes sense, because the handler is at the end of the request pipeline, so there will be no next middleware to invoke.

Migrating handler insertion into the request pipeline

Configuring an HTTP handler is done in *Web.config* and looks something like this:

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <handlers>
      <add name="MyHandler" verb="*" path="*.report"
type="MyApp.HttpHandlers.MyHandler" resourceType="Unspecified"
preCondition="integratedMode"/>
    </handlers>
  </system.webServer>
</configuration>
```

You could convert this by adding your new handler middleware to the request pipeline in your `Startup` class, similar to middleware converted from modules. The problem with that approach is that it would send all requests to your new handler middleware. However, you only want requests with a given extension to reach your middleware. That would give you the same functionality you had with your HTTP handler.

One solution is to branch the pipeline for requests with a given extension, using the `MapWhen` extension method. You do this in the same `Configure` method where you add the other middleware:

C#

```csharp
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

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

    app.UseMyMiddleware();
    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions =
Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptio
ns>();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOpti
ons>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

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

`MapWhen` takes these parameters:

1. A lambda that takes the `HttpContext` and returns `true` if the request should go down the branch. This means you can branch requests not just based on their extension, but also on request headers, query string parameters, etc.

2. A lambda that takes an `IApplicationBuilder` and adds all the middleware for the branch. This means you can add additional middleware to the branch in front of your handler middleware.

Middleware added to the pipeline before the branch will be invoked on all requests; the branch will have no impact on them.

Loading middleware options using the options pattern

Some modules and handlers have configuration options that are stored in *Web.config*. However, in ASP.NET Core a new configuration model is used in place of *Web.config*.

The new configuration system gives you these options to solve this:

- Directly inject the options into the middleware, as shown in the next section.
- Use the options pattern:

1. Create a class to hold your middleware options, for example:

   C#

   ```csharp
   public class MyMiddlewareOptions
   {
       public string Param1 { get; set; }
       public string Param2 { get; set; }
   }
   ```

2. Store the option values

   The configuration system allows you to store option values anywhere you want. However, most sites use `appsettings.json`, so we'll take that approach:

   JSON

   ```json
   {
     "MyMiddlewareOptionsSection": {
       "Param1": "Param1Value",
       "Param2": "Param2Value"
     }
   }
   ```

   MyMiddlewareOptionsSection here is a section name. It doesn't have to be the same as the name of your options class.

3. Associate the option values with the options class

   The options pattern uses ASP.NET Core's dependency injection framework to associate the options type (such as `MyMiddlewareOptions`) with a `MyMiddlewareOptions` object that has the actual options.

   Update your `Startup` class:

a. If you're using `appsettings.json`, add it to the configuration builder in the `Startup` constructor:

```csharp
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true,
reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

b. Configure the options service:

```csharp
public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
    services.AddMvc();
}
```

c. Associate your options with your options class:

```csharp
public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
```

```
        services.AddMvc();
    }
```

4. Inject the options into your middleware constructor. This is similar to injecting options into a controller.

```C#
public class MyMiddlewareWithParams
{
    private readonly RequestDelegate _next;
    private readonly MyMiddlewareOptions _myMiddlewareOptions;

    public MyMiddlewareWithParams(RequestDelegate next,
        IOptions<MyMiddlewareOptions> optionsAccessor)
    {
        _next = next;
        _myMiddlewareOptions = optionsAccessor.Value;
    }

    public async Task Invoke(HttpContext context)
    {
        // Do something with context near the beginning of request
processing
        // using configuration in _myMiddlewareOptions

        await _next.Invoke(context);

        // Do something with context near the end of request processing
        // using configuration in _myMiddlewareOptions
    }
}
```

The UseMiddleware extension method that adds your middleware to the `IApplicationBuilder` takes care of dependency injection.

This isn't limited to `IOptions` objects. Any other object that your middleware requires can be injected this way.

Loading middleware options through direct injection

The options pattern has the advantage that it creates loose coupling between options values and their consumers. Once you've associated an options class with the actual options values, any other class can get access to the options through the dependency injection framework. There's no need to pass around options values.

This breaks down though if you want to use the same middleware twice, with different options. For example an authorization middleware used in different branches allowing different roles. You can't associate two different options objects with the one options class.

The solution is to get the options objects with the actual options values in your `Startup` class and pass those directly to each instance of your middleware.

1. Add a second key to `appsettings.json`

   To add a second set of options to the `appsettings.json` file, use a new key to uniquely identify it:

   JSON

   ```json
   {
     "MyMiddlewareOptionsSection2": {
       "Param1": "Param1Value2",
       "Param2": "Param2Value2"
     },
     "MyMiddlewareOptionsSection": {
       "Param1": "Param1Value",
       "Param2": "Param2Value"
     }
   }
   ```

2. Retrieve options values and pass them to middleware. The `Use...` extension method (which adds your middleware to the pipeline) is a logical place to pass in the option values:

   C#

   ```csharp
   public void Configure(IApplicationBuilder app, IHostingEnvironment env,
   ILoggerFactory loggerFactory)
   {
       loggerFactory.AddConsole(Configuration.GetSection("Logging"));
       loggerFactory.AddDebug();

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

       app.UseMyMiddleware();
   ```

```
    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions =
Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddleware
Options>();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewar
eOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context =>
context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

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

3. Enable middleware to take an options parameter. Provide an overload of the `Use...` extension method (that takes the options parameter and passes it to `UseMiddleware`). When `UseMiddleware` is called with parameters, it passes the parameters to your middleware constructor when it instantiates the middleware object.

C#

```csharp
public static class MyMiddlewareWithParamsExtensions
{
    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>();
    }

    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder, MyMiddlewareOptions
myMiddlewareOptions)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>(
            new OptionsWrapper<MyMiddlewareOptions>
(myMiddlewareOptions));
    }
}
```

Note how this wraps the options object in an `OptionsWrapper` object. This implements `IOptions`, as expected by the middleware constructor.

Migrating to the new HttpContext

You saw earlier that the `Invoke` method in your middleware takes a parameter of type `HttpContext`:

```C#
public async Task Invoke(HttpContext context)
```

`HttpContext` has significantly changed in ASP.NET Core. This section shows how to translate the most commonly used properties of System.Web.HttpContext to the new `Microsoft.AspNetCore.Http.HttpContext`.

HttpContext

HttpContext.Items translates to:

```C#
IDictionary<object, object> items = httpContext.Items;
```

Unique request ID (no System.Web.HttpContext counterpart)

Gives you a unique id for each request. Very useful to include in your logs.

C#

```
string requestId = httpContext.TraceIdentifier;
```

HttpContext.Request

HttpContext.Request.HttpMethod translates to:

C#

```
string httpMethod = httpContext.Request.Method;
```

HttpContext.Request.QueryString translates to:

C#

```
IQueryCollection queryParameters = httpContext.Request.Query;

// If no query parameter "key" used, values will have 0 items
// If single value used for a key (...?key=v1), values will have 1 item
("v1")
// If key has multiple values (...?key=v1&key=v2), values will have 2 items
("v1" and "v2")
IList<string> values = queryParameters["key"];

// If no query parameter "key" used, value will be ""
// If single value used for a key (...?key=v1), value will be "v1"
// If key has multiple values (...?key=v1&key=v2), value will be "v1,v2"
string value = queryParameters["key"].ToString();
```

HttpContext.Request.Url and **HttpContext.Request.RawUrl** translate to:

C#

```
// using Microsoft.AspNetCore.Http.Extensions;
var url = httpContext.Request.GetDisplayUrl();
```

HttpContext.Request.IsSecureConnection translates to:

C#

```
var isSecureConnection = httpContext.Request.IsHttps;
```

HttpContext.Request.UserHostAddress translates to:

C#

```
var userHostAddress = httpContext.Connection.RemoteIpAddress?.ToString();
```

HttpContext.Request.Cookies translates to:

```C#
IRequestCookieCollection cookies = httpContext.Request.Cookies;
string unknownCookieValue = cookies["unknownCookie"]; // will be null (no
exception)
string knownCookieValue = cookies["cookie1name"];     // will be actual
value
```

HttpContext.Request.RequestContext.RouteData translates to:

```C#
var routeValue = httpContext.GetRouteValue("key");
```

HttpContext.Request.Headers translates to:

```C#
// using Microsoft.AspNetCore.Http.Headers;
// using Microsoft.Net.Http.Headers;

IHeaderDictionary headersDictionary = httpContext.Request.Headers;

// GetTypedHeaders extension method provides strongly typed access to many
headers
var requestHeaders = httpContext.Request.GetTypedHeaders();
CacheControlHeaderValue cacheControlHeaderValue =
requestHeaders.CacheControl;

// For unknown header, unknownheaderValues has zero items and
unknownheaderValue is ""
IList<string> unknownheaderValues = headersDictionary["unknownheader"];
string unknownheaderValue = headersDictionary["unknownheader"].ToString();

// For known header, knownheaderValues has 1 item and knownheaderValue is
the value
IList<string> knownheaderValues =
headersDictionary[HeaderNames.AcceptLanguage];
string knownheaderValue =
headersDictionary[HeaderNames.AcceptLanguage].ToString();
```

HttpContext.Request.UserAgent translates to:

C#

```
string userAgent = headersDictionary[HeaderNames.UserAgent].ToString();
```

HttpContext.Request.UrlReferrer translates to:

```C#
string urlReferrer = headersDictionary[HeaderNames.Referer].ToString();
```

HttpContext.Request.ContentType translates to:

```C#
// using Microsoft.Net.Http.Headers;

MediaTypeHeaderValue mediaHeaderValue = requestHeaders.ContentType;
string contentType = mediaHeaderValue?.MediaType.ToString();   // ex.
application/x-www-form-urlencoded
string contentMainType = mediaHeaderValue?.Type.ToString();    // ex.
application
string contentSubType = mediaHeaderValue?.SubType.ToString();  // ex. x-www-
form-urlencoded

System.Text.Encoding requestEncoding = mediaHeaderValue?.Encoding;
```

HttpContext.Request.Form translates to:

```C#
if (httpContext.Request.HasFormContentType)
{
    IFormCollection form;

    form = httpContext.Request.Form; // sync
    // Or
    form = await httpContext.Request.ReadFormAsync(); // async

    string firstName = form["firstname"];
    string lastName = form["lastname"];
}
```

⚠ **Warning**

Read form values only if the content sub type is *x-www-form-urlencoded* or *form-data*.

HttpContext.Request.InputStream translates to:

C#

```csharp
string inputBody;
using (var reader = new System.IO.StreamReader(
    httpContext.Request.Body, System.Text.Encoding.UTF8))
{
    inputBody = reader.ReadToEnd();
}
```

⚠ Warning

Use this code only in a handler type middleware, at the end of a pipeline.

You can read the raw body as shown above only once per request. Middleware trying to read the body after the first read will read an empty body.

This doesn't apply to reading a form as shown earlier, because that's done from a buffer.

HttpContext.Response

HttpContext.Response.Status and **HttpContext.Response.StatusDescription** translate to:

C#

```csharp
// using Microsoft.AspNetCore.Http;
httpContext.Response.StatusCode = StatusCodes.Status200OK;
```

HttpContext.Response.ContentEncoding and **HttpContext.Response.ContentType** translate to:

C#

```csharp
// using Microsoft.Net.Http.Headers;
var mediaType = new MediaTypeHeaderValue("application/json");
mediaType.Encoding = System.Text.Encoding.UTF8;
httpContext.Response.ContentType = mediaType.ToString();
```

HttpContext.Response.ContentType on its own also translates to:

C#

```
httpContext.Response.ContentType = "text/html";
```

HttpContext.Response.Output translates to:

```C#
string responseContent = GetResponseContent();
await httpContext.Response.WriteAsync(responseContent);
```

HttpContext.Response.TransmitFile

Serving up a file is discussed in Request Features in ASP.NET Core.

HttpContext.Response.Headers

Sending response headers is complicated by the fact that if you set them after anything has been written to the response body, they will not be sent.

The solution is to set a callback method that will be called right before writing to the response starts. This is best done at the start of the `Invoke` method in your middleware. It's this callback method that sets your response headers.

The following code sets a callback method called `SetHeaders`:

```C#
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

The `SetHeaders` callback method would look like this:

```C#
// using Microsoft.AspNet.Http.Headers;
// using Microsoft.Net.Http.Headers;

private Task SetHeaders(object context)
{
    var httpContext = (HttpContext)context;

    // Set header with single value
    httpContext.Response.Headers["ResponseHeaderName"] = "headerValue";

    // Set header with multiple values
    string[] responseHeaderValues = new string[] { "headerValue1",
"headerValue1" };
    httpContext.Response.Headers["ResponseHeaderName"] =
responseHeaderValues;

    // Translating ASP.NET 4's HttpContext.Response.RedirectLocation
    httpContext.Response.Headers[HeaderNames.Location] =
"http://www.example.com";
    // Or
    httpContext.Response.Redirect("http://www.example.com");

    // GetTypedHeaders extension method provides strongly typed access to
many headers
    var responseHeaders = httpContext.Response.GetTypedHeaders();

    // Translating ASP.NET 4's HttpContext.Response.CacheControl
    responseHeaders.CacheControl = new CacheControlHeaderValue
    {
        MaxAge = new System.TimeSpan(365, 0, 0, 0)
        // Many more properties available
    };

    // If you use .NET Framework 4.6+, Task.CompletedTask will be a bit
faster
    return Task.FromResult(0);
}
```

HttpContext.Response.Cookies

Cookies travel to the browser in a *Set-Cookie* response header. As a result, sending cookies requires the same callback as used for sending response headers:

C#

```
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetCookies, state: httpContext);
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

The `SetCookies` callback method would look like the following:

C#

```
private Task SetCookies(object context)
{
    var httpContext = (HttpContext)context;

    IResponseCookies responseCookies = httpContext.Response.Cookies;

    responseCookies.Append("cookie1name", "cookie1value");
    responseCookies.Append("cookie2name", "cookie2value",
        new CookieOptions { Expires = System.DateTime.Now.AddDays(5),
HttpOnly = true });

    // If you use .NET Framework 4.6+, Task.CompletedTask will be a bit
faster
    return Task.FromResult(0);
}
```

Additional resources

- Incremental HTTP module migration
- HTTP Handlers and HTTP Modules Overview
- Configuration
- Application Startup
- Middleware