

```

        $"Queued Hosted Service is running.{Environment.NewLine}" +
        $"{Environment.NewLine}Tap W to add a work item to the " +
        $"background queue.{Environment.NewLine}");

        await BackgroundProcessing(stoppingToken);
    }

    private async Task BackgroundProcessing(Cancellation_token stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            var workItem =
                await TaskQueue.DequeueAsync(stoppingToken);

            try
            {
                await workItem(stoppingToken);
            }
            catch (Exception ex)
            {
                _logger.LogError(ex,
                    "Error occurred executing {WorkItem}.",
                    nameof(workItem));
            }
        }
    }

    public override async Task StopAsync(Cancellation_token stoppingToken)
    {
        _logger.LogInformation("Queued Hosted Service is stopping.");

        await base.StopAsync(stoppingToken);
    }
}

```

A `MonitorLoop` service handles enqueueing tasks for the hosted service whenever the `w` key is selected on an input device:

- The `IBackgroundTaskQueue` is injected into the `MonitorLoop` service.
- `IBackgroundTaskQueue.QueueBackgroundWorkItem` is called to enqueue a work item.
- The work item simulates a long-running background task:
 - Three 5-second delays are executed (`Task.Delay`).
 - A `try-catch` statement traps `OperationCanceledException` if the task is cancelled.

C#

```

public class MonitorLoop
{
    private readonly IBackgroundTaskQueue _taskQueue;

```

```

private readonly ILogger _logger;
private readonly CancellationToken _cancellationToken;

public MonitorLoop(IBackgroundTaskQueue taskQueue,
    ILogger<MonitorLoop> logger,
    IHostApplicationLifetime applicationLifetime)
{
    _taskQueue = taskQueue;
    _logger = logger;
    _cancellationToken = applicationLifetime.ApplicationStopping;
}

public void StartMonitorLoop()
{
    _logger.LogInformation("MonitorAsync Loop is starting.");

    // Run a console user input loop in a background thread
    Task.Run(async () => await MonitorAsync());
}

private async ValueTask MonitorAsync()
{
    while (!_cancellationToken.IsCancellationRequested)
    {
        var keyStroke = Console.ReadKey();

        if (keyStroke.Key == ConsoleKey.W)
        {
            // Enqueue a background work item
            await
                _taskQueue.QueueBackgroundWorkItemAsync(BuildWorkItem);
        }
    }
}

private async ValueTask BuildWorkItem(CancellationToken token)
{
    // Simulate three 5-second tasks to complete
    // for each enqueued work item

    int delayLoop = 0;
    var guid = Guid.NewGuid().ToString();

    _logger.LogInformation("Queued Background Task {Guid} is starting.",
guid);

    while (!token.IsCancellationRequested && delayLoop < 3)
    {
        try
        {
            await Task.Delay(TimeSpan.FromSeconds(5), token);
        }
        catch (OperationCanceledException)
        {
            // Prevent throwing if the Delay is cancelled

```

```

    }

    delayLoop++;

    _logger.LogInformation("Queued Background Task {Guid} is
running. "
                           + "{DelayLoop}/3", guid, delayLoop);
    }

    if (delayLoop == 3)
    {
        _logger.LogInformation("Queued Background Task {Guid} is
complete.", guid);
    }
    else
    {
        _logger.LogInformation("Queued Background Task {Guid} was
cancelled.", guid);
    }
}
}

```

The services are registered in `IHostBuilder.ConfigureServices` (`Program.cs`). The hosted service is registered with the `AddHostedService` extension method:

C#

```

services.AddSingleton<MonitorLoop>();
services.AddHostedService<QueuedHostedService>();
services.AddSingleton<IBackgroundTaskQueue>(ctx =>
{
    if (!int.TryParse(hostContext.Configuration["QueueCapacity"], out var
queueCapacity))
        queueCapacity = 100;
    return new BackgroundTaskQueue(queueCapacity);
});

```

`MonitorLoop` is started in `Program.cs`:

C#

```

var monitorLoop = host.Services.GetRequiredService<MonitorLoop>();
monitorLoop.StartMonitorLoop();

```

Asynchronous timed background task

The following code creates an asynchronous timed background task:

C#

```
namespace TimedBackgroundTasks;

public class TimedHostedService : BackgroundService
{
    private readonly ILogger<TimedHostedService> _logger;
    private int _executionCount;

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

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        _logger.LogInformation("Timed Hosted Service running.");

        // When the timer should have no due-time, then do the work once
now.
        DoWork();

        using PeriodicTimer timer = new(TimeSpan.FromSeconds(1));

        try
        {
            while (await timer.WaitForNextTickAsync(stoppingToken))
            {
                DoWork();
            }
        }
        catch (OperationCanceledException)
        {
            _logger.LogInformation("Timed Hosted Service is stopping.");
        }
    }

    // Could also be a async method, that can be awaited in ExecuteAsync
above
    private void DoWork()
    {
        int count = Interlocked.Increment(ref _executionCount);

        _logger.LogInformation("Timed Hosted Service is working. Count:
{Count}", count);
    }
}
```

Native AOT

The Worker Service templates support [.NET native ahead-of-time \(AOT\)](#) with the `--aot` flag:

Visual Studio

1. Create a new project.
2. Select **Worker Service**. Select **Next**.
3. Provide a project name in the **Project name** field or accept the default project name. Select **Next**.
4. In the **Additional information** dialog:
5. Choose a **Framework**.
6. Check the **Enable Native AOT publish** checkbox.
7. Select **Create**.

The AOT option adds `<PublishAot>true</PublishAot>` to the project file:

diff

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <InvariantGlobalization>true</InvariantGlobalization>
+   <PublishAot>true</PublishAot>
    <UserSecretsId>dotnet-WorkerWithAot-e94b2</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0-
    preview.4.23259.5" />
  </ItemGroup>
</Project>
```

Additional resources

- [Background services unit tests on GitHub](#) [↗].
- [View or download sample code](#) [↗] (how to download)
- [Implement background tasks in microservices with IHostedService and the BackgroundService class](#)
- [Run background tasks with WebJobs in Azure App Service](#)

- Timer

Use hosting startup assemblies in ASP.NET Core

Article • 07/26/2024

📘 Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

By [Pavel Krymets](#) 

An [IHostingStartup](#) (hosting startup) implementation adds enhancements to an app at startup from an external assembly. For example, an external library can use a hosting startup implementation to provide additional configuration providers or services to an app.

[View or download sample code](#)  ([how to download](#))

HostingStartup attribute

A [HostingStartup](#) attribute indicates the presence of a hosting startup assembly to activate at runtime.

The entry assembly or the assembly containing the `Startup` class is automatically scanned for the `HostingStartup` attribute. The list of assemblies to search for `HostingStartup` attributes is loaded at runtime from configuration in the [WebHostDefaults.HostingStartupAssembliesKey](#). The list of assemblies to exclude from discovery is loaded from the [WebHostDefaults.HostingStartupExcludeAssembliesKey](#).

In the following example, the namespace of the hosting startup assembly is `StartupEnhancement`. The class containing the hosting startup code is `StartupEnhancementHostingStartup`:

C#

```
[assembly: HostingStartup(typeof(StartupEnhancement.StartupEnhancementHostingStartup))]
```

The `HostingStartup` attribute is typically located in the hosting startup assembly's `IHostingStartup` implementation class file.

Discover loaded hosting startup assemblies

To discover loaded hosting startup assemblies, enable logging and check the app's logs. Errors that occur when loading assemblies are logged. Loaded hosting startup assemblies are logged at the Debug level, and all errors are logged.

Disable automatic loading of hosting startup assemblies

To disable automatic loading of hosting startup assemblies, use one of the following approaches:

- To prevent all hosting startup assemblies from loading, set one of the following to `true` or `1`:
 - Prevent Hosting Startup host configuration setting:

C#

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

- `ASPNETCORE_PREVENTHOSTINGSTARTUP` environment variable.
- To prevent specific hosting startup assemblies from loading, set one of the following to a semicolon-delimited string of hosting startup assemblies to exclude at startup:
 - Hosting Startup Exclude Assemblies host configuration setting:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseSetting(
                WebHostDefaults.HostingStartupExcludeAssembliesKey,
                "{ASSEMBLY1;ASSEMBLY2; ...}")
                .UseStartup<Startup>();
        });
```

The `{ASSEMBLY1;ASSEMBLY2; ...}` placeholder represents the semicolon-separated list of assemblies.

- `ASPNETCORE_HOSTINGSTARTUPEXCLUDEASSEMBLIES` environment variable.

If both the host configuration setting and the environment variable are set, the host setting controls the behavior.

Disabling hosting startup assemblies using the host setting or environment variable disables the assembly globally and may disable several characteristics of an app.

Project

Create a hosting startup with either of the following project types:

- [Class library](#)
- [Console app without an entry point](#)

Class library

A hosting startup enhancement can be provided in a class library. The library contains a `HostingStartup` attribute.

The [sample code](#) includes a Razor Pages app, *HostingStartupApp*, and a class library, *HostingStartupLibrary*. The class library:

- Contains a hosting startup class, `ServiceKeyInjection`, which implements `IHostingStartup`. `ServiceKeyInjection` adds a pair of service strings to the app's configuration using the in-memory configuration provider ([AddInMemoryCollection](#)).
- Includes a `HostingStartup` attribute that identifies the hosting startup's namespace and class.

The `ServiceKeyInjection` class's [Configure](#) method uses an [IWebHostBuilder](#) to add enhancements to an app.

`HostingStartupLibrary/ServiceKeyInjection.cs`:

```
C#

[assembly: HostingStartup(typeof(HostingStartupLibrary.ServiceKeyInjection))]

namespace HostingStartupLibrary
{
    public class ServiceKeyInjection : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureAppConfiguration(config =>
            {
                var dict = new Dictionary<string, string>
                {
                    {"DevAccount_FromLibrary", "DEV_111111-1111"},
                    {"ProdAccount_FromLibrary", "PROD_222222-2222"}
                };

                config.AddInMemoryCollection(dict);
            });
        }
    }
}
```

The app's Index page reads and renders the configuration values for the two keys set by the class library's hosting startup assembly:

HostingStartupApp/Pages/Index.cshtml.cs:

```
C#

public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        ServiceKey_Development_Library = config["DevAccount_FromLibrary"];
        ServiceKey_Production_Library = config["ProdAccount_FromLibrary"];
        ServiceKey_Development_Package = config["DevAccount_FromPackage"];
        ServiceKey_Production_Package = config["ProdAccount_FromPackage"];
    }

    public string ServiceKey_Development_Library { get; private set; }
    public string ServiceKey_Production_Library { get; private set; }
    public string ServiceKey_Development_Package { get; private set; }
    public string ServiceKey_Production_Package { get; private set; }

    public void OnGet()
    {
    }
}
```

The [sample code](#) also includes a NuGet package project that provides a separate hosting startup, *HostingStartupPackage*. The package has the same characteristics of the class library described earlier. The package:

- Contains a hosting startup class, `ServiceKeyInjection`, which implements `IHostingStartup`. `ServiceKeyInjection` adds a pair of service strings to the app's configuration.
- Includes a `HostingStartup` attribute.

HostingStartupPackage/ServiceKeyInjection.cs:

```
C#

[assembly: HostingStartup(typeof(HostingStartupPackage.ServiceKeyInjection))]

namespace HostingStartupPackage
{
    public class ServiceKeyInjection : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureAppConfiguration(config =>
            {
                var dict = new Dictionary<string, string>
                {
                    {"DevAccount_FromPackage", "DEV_3333333-3333"},
                    {"ProdAccount_FromPackage", "PROD_4444444-4444"}
                };

                config.AddInMemoryCollection(dict);
            });
        }
    }
}
```

```

    });
}
}
}

```

The app's Index page reads and renders the configuration values for the two keys set by the package's hosting startup assembly:

HostingStartupApp/Pages/Index.cshtml.cs:

C#

```

public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        ServiceKey_Development_Library = config["DevAccount_FromLibrary"];
        ServiceKey_Production_Library = config["ProdAccount_FromLibrary"];
        ServiceKey_Development_Package = config["DevAccount_FromPackage"];
        ServiceKey_Production_Package = config["ProdAccount_FromPackage"];
    }

    public string ServiceKey_Development_Library { get; private set; }
    public string ServiceKey_Production_Library { get; private set; }
    public string ServiceKey_Development_Package { get; private set; }
    public string ServiceKey_Production_Package { get; private set; }

    public void OnGet()
    {
    }
}

```

Console app without an entry point

This approach is only available for .NET Core apps, not .NET Framework.

A dynamic hosting startup enhancement that doesn't require a compile-time reference for activation can be provided in a console app without an entry point that contains a `HostingStartup` attribute. Publishing the console app produces a hosting startup assembly that can be consumed from the runtime store.

A console app without an entry point is used in this process because:

- A dependencies file is required to consume the hosting startup in the hosting startup assembly. A dependencies file is a runnable app asset that's produced by publishing an app, not a library.
- A library can't be added directly to the [runtime package store](#), which requires a runnable project that targets the shared runtime.

In the creation of a dynamic hosting startup:

- A hosting startup assembly is created from the console app without an entry point that:
 - Includes a class that contains the `IHostingStartup` implementation.

- Includes a [HostingStartup](#) attribute to identify the `IHostingStartup` implementation class.
- The console app is published to obtain the hosting startup's dependencies. A consequence of publishing the console app is that unused dependencies are trimmed from the dependencies file.
- The dependencies file is modified to set the runtime location of the hosting startup assembly.
- The hosting startup assembly and its dependencies file is placed into the runtime package store. To discover the hosting startup assembly and its dependencies file, they're listed in a pair of environment variables.

The console app references the [Microsoft.AspNetCore.Hosting.Abstractions](#) package:

```
XML

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Hosting.Abstractions"
                      Version="3.0.0" />
  </ItemGroup>

</Project>
```

A [HostingStartup](#) attribute identifies a class as an implementation of `IHostingStartup` for loading and execution when building the [IWebHost](#). In the following example, the namespace is `StartupEnhancement`, and the class is `StartupEnhancementHostingStartup`:

```
C#

[assembly: HostingStartup(typeof(StartupEnhancement.StartupEnhancementHostingStartup))]
```

A class implements `IHostingStartup`. The class's [Configure](#) method uses an [IWebHostBuilder](#) to add enhancements to an app. `IHostingStartup.Configure` in the hosting startup assembly is called by the runtime before `Startup.Configure` in user code, which allows user code to overwrite any configuration provided by the hosting startup assembly.

```
C#

namespace StartupEnhancement
{
    public class StartupEnhancementHostingStartup : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            // Use the IWebHostBuilder to add app enhancements.
        }
    }
}
```

When building an `IHostingStartup` project, the dependencies file (`.deps.json`) sets the `runtime` location of the assembly to the `bin` folder:

JSON

```
"targets": {
  ".NETCoreApp,Version=v3.0": {
    "StartupEnhancement/1.0.0": {
      "dependencies": {
        "Microsoft.AspNetCore.Hosting.Abstractions": "3.0.0"
      },
      "runtime": {
        "StartupEnhancement.dll": {}
      }
    }
  }
}
```

Only part of the file is shown. The assembly name in the example is `StartupEnhancement`.

Configuration provided by the hosting startup

There are two approaches to handling configuration depending on whether you want the hosting startup's configuration to take precedence or the app's configuration to take precedence:

1. Provide configuration to the app using [ConfigureAppConfiguration](#) to load the configuration after the app's [ConfigureAppConfiguration](#) delegates execute. Hosting startup configuration takes priority over the app's configuration using this approach.
2. Provide configuration to the app using [UseConfiguration](#) to load the configuration before the app's [ConfigureAppConfiguration](#) delegates execute. The app's configuration values take priority over those provided by the hosting startup using this approach.

C#

```
public class ConfigurationInjection : IHostingStartup
{
    public void Configure(IWebHostBuilder builder)
    {
        Dictionary<string, string> dict;

        builder.ConfigureAppConfiguration(config =>
        {
            dict = new Dictionary<string, string>
            {
                {"ConfigurationKey1",
                 "From IHostingStartup: Higher priority " +
                 "than the app's configuration."},
            };

            config.AddInMemoryCollection(dict);
        });

        dict = new Dictionary<string, string>
        {
```

```

        {"ConfigurationKey2",
         "From IHostingStartup: Lower priority " +
         "than the app's configuration."},
    };

    var builtConfig = new ConfigurationBuilder()
        .AddInMemoryCollection(dict)
        .Build();

    builder.UseConfiguration(builtConfig);
}
}

```

Specify the hosting startup assembly

For either a class library- or console app-supplied hosting startup, specify the hosting startup assembly's name in the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable. The environment variable is a semicolon-delimited list of assemblies.

Only hosting startup assemblies are scanned for the `HostingStartup` attribute. For the sample app, *HostingStartupApp*, to discover the hosting startups described earlier, the environment variable is set to the following value:

```
HostingStartupLibrary;HostingStartupPackage;StartupDiagnostics
```

A hosting startup assembly can also be set using the Hosting Startup Assemblies host configuration setting:

```

C#

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseSetting(
                WebHostDefaults.HostingStartupAssembliesKey,
                "{ASSEMBLY1;ASSEMBLY2; ...}")
            .UseStartup<Startup>();
        });

```

The `{ASSEMBLY1;ASSEMBLY2; ...}` placeholder represents the semicolon-separated list of assemblies.

When multiple hosting startup assemblies are present, their `Configure` methods are executed in the order that the assemblies are listed.

Activation

Options for hosting startup activation are:

- **Runtime store:** Activation doesn't require a compile-time reference for activation. The sample app places the hosting startup assembly and dependencies files into a folder, *deployment*, to facilitate deployment of the hosting startup in a multimachine environment. The *deployment* folder also includes a PowerShell script that creates or modifies environment variables on the deployment system to enable the hosting startup.
- Compile-time reference required for activation
 - [NuGet package](#)
 - [Project bin folder](#)

Runtime store

The hosting startup implementation is placed in the [runtime store](#). A compile-time reference to the assembly isn't required by the enhanced app.

After the hosting startup is built, a runtime store is generated using the manifest project file and the [dotnet store](#) command.

.NET CLI

```
dotnet store --manifest {MANIFEST FILE} --runtime {RUNTIME IDENTIFIER} --output {OUTPUT LOCATION} --skip-optimization
```

In the sample app (*RuntimeStore* project) the following command is used:

.NET CLI

```
dotnet store --manifest store.manifest.csproj --runtime win7-x64 --output ./deployment/store --skip-optimization
```

For the runtime to discover the runtime store, the runtime store's location is added to the `DOTNET_SHARED_STORE` environment variable.

Modify and place the hosting startup's dependencies file

To activate the enhancement without a package reference to the enhancement, specify additional dependencies to the runtime with `additionalDeps`. `additionalDeps` allows you to:

- Extend the app's library graph by providing a set of additional `.deps.json` files to merge with the app's own `.deps.json` file on startup.
- Make the hosting startup assembly discoverable and loadable.

The recommended approach for generating the additional dependencies file is to:

1. Execute `dotnet publish` on the runtime store manifest file referenced in the previous section.
2. Remove the manifest reference from libraries and the `runtime` section of the resulting `.deps.json` file.

In the example project, the `store.manifest/1.0.0` property is removed from the `targets` and `libraries` section:

JSON

```
{
  "runtimeTarget": {
    "name": ".NETCoreApp,Version=v3.0",
    "signature": ""
  },
  "compilationOptions": {},
  "targets": {
    ".NETCoreApp,Version=v3.0": {
      "store.manifest/1.0.0": {
        "dependencies": {
          "StartupDiagnostics": "1.0.0"
        },
        "runtime": {
          "store.manifest.dll": {}
        }
      },
      "StartupDiagnostics/1.0.0": {
        "runtime": {
          "lib/netcoreapp3.0/StartupDiagnostics.dll": {
            "assemblyVersion": "1.0.0.0",
            "fileVersion": "1.0.0.0"
          }
        }
      }
    }
  },
  "libraries": {
    "store.manifest/1.0.0": {
      "type": "project",
      "serviceable": false,
      "sha512": ""
    },
    "StartupDiagnostics/1.0.0": {
      "type": "package",
      "serviceable": true,
      "sha512": "sha512-
xrhzuNSyM5/f4ZswhooJ9dmIYLP64wMnqUJSyTKVDKDVj5T+qtzyp18JmM/aFJLLpYrF0FYpVWvGujd7/FfMEw=
",
      "path": "startupdiagnostics/1.0.0",
      "hashPath": "startupdiagnostics.1.0.0.nupkg.sha512"
    }
  }
}
```

Place the `.deps.json` file into the following location:

```
{ADDITIONAL DEPENDENCIES PATH}/shared/{SHARED FRAMEWORK NAME}/{SHARED FRAMEWORK
VERSION}/{ENHANCEMENT ASSEMBLY NAME}.deps.json
```


- `{ADDITIONAL_DEPENDENCIES_PATH}`: Location added to the `DOTNET_ADDITIONAL_DEPS` environment variable.
- `{SHARED_FRAMEWORK_NAME}`: Shared framework required for this additional dependencies file.
- `{SHARED_FRAMEWORK_VERSION}`: Minimum shared framework version.
- `{ENHANCEMENT_ASSEMBLY_NAME}`: The enhancement's assembly name.

In the sample app (*RuntimeStore* project), the additional dependencies file is placed into the following location:

```
deployment/additionalDeps/shared/Microsoft.AspNetCore.App/3.0.0/StartupDiagnostics.deps.json
```

For runtime to discover the runtime store location, the additional dependencies file location is added to the `DOTNET_ADDITIONAL_DEPS` environment variable.

In the sample app (*RuntimeStore* project), building the runtime store and generating the additional dependencies file is accomplished using a [PowerShell](#) script.

For examples of how to set environment variables for various operating systems, see [Use multiple environments](#).

Deployment

To facilitate the deployment of a hosting startup in a multimachine environment, the sample app creates a *deployment* folder in published output that contains:

- The hosting startup runtime store.
- The hosting startup dependencies file.
- A PowerShell script that creates or modifies the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES`, `DOTNET_SHARED_STORE`, and `DOTNET_ADDITIONAL_DEPS` to support the activation of the hosting startup. Run the script from an administrative PowerShell command prompt on the deployment system.

NuGet package

A hosting startup enhancement can be provided in a NuGet package. The package has a `HostingStartup` attribute. The hosting startup types provided by the package are made available to the app using either of the following approaches:

- The enhanced app's project file makes a package reference for the hosting startup in the app's project file (a compile-time reference). With the compile-time reference in place, the hosting startup assembly and all of its dependencies are incorporated into the app's dependency file (`.deps.json`). This approach applies to a hosting startup assembly package published to nuget.org.
- The hosting startup's dependencies file is made available to the enhanced app as described in the [Runtime store](#) section (without a compile-time reference).

For more information on NuGet packages and the runtime store, see the following topics:

- [How to Create a NuGet Package with Cross Platform Tools](#)
- [Publishing packages](#)
- [Runtime package store](#)

Project bin folder

A hosting startup enhancement can be provided by a *bin*-deployed assembly in the enhanced app. The hosting startup types provided by the assembly are made available to the app using one of the following approaches:

- The enhanced app's project file makes an assembly reference to the hosting startup (a compile-time reference). With the compile-time reference in place, the hosting startup assembly and all of its dependencies are incorporated into the app's dependency file (`.deps.json`). This approach applies when the deployment scenario calls for making a compile-time reference to the hosting startup's assembly (`.dll` file) and moving the assembly to either:
 - The consuming project.
 - A location accessible by the consuming project.
- The hosting startup's dependencies file is made available to the enhanced app as described in the [Runtime store](#) section (without a compile-time reference).
- When targeting the .NET Framework, the assembly is loadable in the default load context, which on .NET Framework means that the assembly is located at either of the following locations:
 - Application base path: The *bin* folder where the app's executable (`.exe`) is located.
 - Global Assembly Cache (GAC): The GAC stores assemblies that several .NET Framework apps share. For more information, see [How to: Install an assembly into the global assembly cache](#) in the .NET Framework documentation.

Sample code

The [sample code](#) [\(how to download\)](#) demonstrates hosting startup implementation scenarios:

- Two hosting startup assemblies (class libraries) set a pair of in-memory configuration key-value pairs each:
 - NuGet package (*HostingStartupPackage*)
 - Class library (*HostingStartupLibrary*)
- A hosting startup is activated from a runtime store-deployed assembly (*StartupDiagnostics*). The assembly adds two middlewares to the app at startup that provide diagnostic information on:
 - Registered services
 - Address (scheme, host, path base, path, query string)
 - Connection (remote IP, remote port, local IP, local port, client certificate)
 - Request headers
 - Environment variables

To run the sample:

Activation from a NuGet package

1. Compile the *HostingStartupPackage* package with the `dotnet pack` command.
2. Add the package's assembly name of the *HostingStartupPackage* to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
3. Compile and run the app. A package reference is present in the enhanced app (a compile-time reference). A `<PropertyGroup>` in the app's project file specifies the package project's output (`./HostingStartupPackage/bin/Debug`) as a package source. This allows the app to use the package without uploading the package to nuget.org. For more information, see the notes in the *HostingStartupApp*'s project file.

XML

```
<PropertyGroup>

<RestoreSources>$(RestoreSources);https://api.nuget.org/v3/index.json;../HostingSt
artupPackage/bin/Debug</RestoreSources>
</PropertyGroup>
```

4. Observe that the service configuration key values rendered by the Index page match the values set by the package's `ServiceKeyInjection.Configure` method.

If you make changes to the *HostingStartupPackage* project and recompile it, clear the local NuGet package caches to ensure that the *HostingStartupApp* receives the updated package and not a stale package from the local cache. To clear the local NuGet caches, execute the following `dotnet nuget locals` command:

.NET CLI

```
dotnet nuget locals all --clear
```

Activation from a class library

1. Compile the *HostingStartupLibrary* class library with the `dotnet build` command.
2. Add the class library's assembly name of *HostingStartupLibrary* to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
3. *bin*-deploy the class library's assembly to the app by copying the *HostingStartupLibrary.dll* file from the class library's compiled output to the app's *bin/Debug* folder.
4. Compile and run the app. An `<ItemGroup>` in the app's project file references the class library's assembly (`./bin/Debug/netcoreapp3.0/HostingStartupLibrary.dll`) (a compile-time reference). For more information, see the notes in the *HostingStartupApp*'s project file.

XML

```
<ItemGroup>
  <Reference Include=".\bin\Debug\netcoreapp3.0\HostingStartupLibrary.dll">
    <HintPath>.\bin\Debug\netcoreapp3.0\HostingStartupLibrary.dll</HintPath>
    <SpecificVersion>False</SpecificVersion>
  </Reference>
</ItemGroup>
```

5. Observe that the service configuration key values rendered by the Index page match the values set by the class library's `ServiceKeyInjection.Configure` method.

Activation from a runtime store-deployed assembly

1. The *StartupDiagnostics* project uses [PowerShell](#) to modify its `StartupDiagnostics.deps.json` file. PowerShell is installed by default on Windows starting with Windows 7 SP1 and Windows Server 2008 R2 SP1. To obtain PowerShell on other platforms, see [Installing various versions of PowerShell](#).
2. Execute the *build.ps1* script in the *RuntimeStore* folder. The script:
 - Generates the `StartupDiagnostics` package in the *obj\packages* folder.
 - Generates the runtime store for `StartupDiagnostics` in the *store* folder. The `dotnet store` command in the script uses the `win7-x64` [runtime identifier \(RID\)](#) for a hosting startup deployed to Windows. When providing the hosting startup for a different runtime, substitute the correct RID on line 37 of the script. The runtime store for `StartupDiagnostics` would later be moved to the user's or system's runtime store on the machine where the assembly will be consumed. The user runtime store install location for the `StartupDiagnostics` assembly is `.dotnet/store/x64/netcoreapp3.0/startupdiagnostics/1.0.0/lib/netcoreapp3.0/StartupDiagnostics.dll`.
 - Generates the `additionalDeps` for `StartupDiagnostics` in the *additionalDeps* folder. The additional dependencies would later be moved to the user's or system's additional dependencies. The user `StartupDiagnostics` additional dependencies install location is `.dotnet/x64/additionalDeps/StartupDiagnostics/shared/Microsoft.NETCore.App/3.0.0/StartupDiagnostics.deps.json`.
 - Places the *deploy.ps1* file in the *deployment* folder.
3. Run the *deploy.ps1* script in the *deployment* folder. The script appends:
 - `StartupDiagnostics` to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
 - The hosting startup dependencies path (in the *RuntimeStore* project's *deployment* folder) to the `DOTNET_ADDITIONAL_DEPS` environment variable.
 - The runtime store path (in the *RuntimeStore* project's *deployment* folder) to the `DOTNET_SHARED_STORE` environment variable.
4. Run the sample app.
5. Request the `/services` endpoint to see the app's registered services. Request the `/diag` endpoint to see the diagnostic information.

Use ASP.NET Core APIs in a class library

Article • 06/18/2024


By [Scott Addie](#) 

This document provides guidance for using ASP.NET Core APIs in a class library. For all other library guidance, see [Open-source library guidance](#).

Determine which ASP.NET Core versions to support

ASP.NET Core adheres to the [.NET Core support policy](#) . Consult the support policy when determining which ASP.NET Core versions to support in a library. A library should:

- Make an effort to support all ASP.NET Core versions classified as *Long-Term Support* (LTS).
- Not feel obligated to support ASP.NET Core versions classified as *End of Life* (EOL).

As preview releases of ASP.NET Core are made available, breaking changes are posted in the [aspnet/Announcements](#)  GitHub repository. Compatibility testing of libraries can be conducted as framework features are being developed.

Use the ASP.NET Core shared framework

With the release of .NET Core 3.0, many ASP.NET Core assemblies are no longer published to NuGet as packages. Instead, the assemblies are included in the `Microsoft.AspNetCore.App` shared framework, which is installed with the .NET Core SDK and runtime installers. For a list of packages no longer being published, see [Remove obsolete package references](#).

As of .NET Core 3.0, projects using the `Microsoft.NET.Sdk.Web` MSBuild SDK implicitly reference the shared framework. Projects using the `Microsoft.NET.Sdk` or `Microsoft.NET.Sdk.Razor` SDK must reference ASP.NET Core to use ASP.NET Core APIs in the shared framework.

To reference ASP.NET Core, add the following `<FrameworkReference>` element to your project file:

```
XML
```

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

Include Blazor extensibility

Blazor supports creating [Razor components](#) class libraries for server-side and client-side apps. To support Razor components in a class library, the class library must use the [Microsoft.NET.Sdk.Razor SDK](#).

Support server-side and client-side apps

To support Razor component consumption by server-side and client-side apps from a single library, use the following instructions for your editor.

Visual Studio

Use the **Razor Class Library** project template.

ⓘ Note

Do **not** select the **Support pages and views** checkbox. Selecting the checkbox results in a class library that only supports server-side apps.

The library generated from the project template:

- Targets the current .NET framework based on the installed SDK.
- Enables browser compatibility checks for platform dependencies by including `browser` as a supported platform with the `SupportedPlatform` MSBuild item.
- Adds a NuGet package reference for [Microsoft.AspNetCore.Components.Web](#) [↗](#).

[RazorClassLibrary-CSharp.csproj \(reference source\)](#) [↗](#)

ⓘ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#). ↗.

Support multiple framework versions

If the library must support features added to Blazor in the current release while also supporting one or more earlier releases, multi-target the library. Provide a semicolon-separated list of [Target Framework Monikers \(TFMs\)](#) in the `TargetFrameworks` MSBuild property:

XML

```
<TargetFrameworks>{TARGET_FRAMEWORKS}</TargetFrameworks>
```

In the preceding example, the `{TARGET_FRAMEWORKS}` placeholder represents the semicolon-separated TFMs list. For example, `netcoreapp3.1;net5.0`.

Only support server-side consumption

Class libraries are rarely built to only support server-side apps. If the class library only requires server-side-specific features, such as access to [CircuitHandler](#) or [Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage](#), or uses ASP.NET Core-specific features, such as middleware, MVC controllers, or Razor Pages, use **one** of the following approaches:

- Specify that the library supports pages and views when the library is created with the **Support pages and views** checkbox (Visual Studio) or the `-s|--support-pages-and-views` option with the `dotnet new` command:

.NET CLI

```
dotnet new razorclasslib -s
```

- Only provide a framework reference to ASP.NET Core in the library's project file in addition to any other required MSBuild properties:

XML

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

For more information on libraries containing Razor components, see [Consume ASP.NET Core Razor components from a Razor class library \(RCL\)](#).

Include MVC extensibility

This section outlines recommendations for libraries that include:

- [Razor views or Razor Pages](#)
- [Tag Helpers](#)
- [View components](#)

This section doesn't discuss multi-targeting to support multiple versions of MVC. For guidance on supporting multiple ASP.NET Core versions, see [Support multiple ASP.NET Core versions](#).

Razor views or Razor Pages

A project that includes [Razor views](#) or [Razor Pages](#) must use the [Microsoft.NET.Sdk.Razor SDK](#).

If the project targets .NET Core 3.x, it requires:

- An `AddRazorSupportForMvc` MSBuild property set to `true`.
- A `<FrameworkReference>` element for the shared framework.

The **Razor Class Library** project template satisfies the preceding requirements for projects targeting .NET Core. Use the following instructions for your editor.

Visual Studio

Use the **Razor Class Library** project template. The template's **Support pages and views** checkbox should be selected.

For example:

XML


```

<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>

```

If the project targets .NET Standard instead, a [Microsoft.AspNetCore.Mvc](#) package reference is required. The `Microsoft.AspNetCore.Mvc` package moved into the shared framework in ASP.NET Core 3.0 and is therefore no longer published. For example:

XML

```

<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  </ItemGroup>

</Project>

```

Tag Helpers

A project that includes [Tag Helpers](#) should use the `Microsoft.NET.Sdk` SDK. If targeting .NET Core 3.x, add a `<FrameworkReference>` element for the shared framework. For example:

XML

```

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

```

```
</Project>
```

If targeting .NET Standard (to support versions earlier than ASP.NET Core 3.x), add a package reference to [Microsoft.AspNetCore.Mvc.Razor](#). The `Microsoft.AspNetCore.Mvc.Razor` package moved into the shared framework and is therefore no longer published. For example:

XML

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

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  </ItemGroup>

</Project>
```

View components

A project that includes [View components](#) should use the `Microsoft.NET.Sdk` SDK. If targeting .NET Core 3.x, add a `<FrameworkReference>` element for the shared framework. For example:

XML

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

If targeting .NET Standard (to support versions earlier than ASP.NET Core 3.x), add a package reference to [Microsoft.AspNetCore.Mvc.ViewFeatures](#). The `Microsoft.AspNetCore.Mvc.ViewFeatures` package moved into the shared framework and is therefore no longer published. For example:

XML

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

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.ViewFeatures"
Version="2.2.0" />
  </ItemGroup>

</Project>
```

Support multiple ASP.NET Core versions

Multi-targeting is required to author a library that supports multiple variants of ASP.NET Core. Consider a scenario in which a Tag Helpers library must support the following ASP.NET Core variants:

- ASP.NET Core 2.1 targeting .NET Framework 4.6.1
- ASP.NET Core 2.x targeting .NET Core 2.x
- ASP.NET Core 3.x targeting .NET Core 3.x

The following project file supports these variants via the `TargetFrameworks` property:

XML

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

  <PropertyGroup>
    <TargetFrameworks>netcoreapp2.1;netcoreapp3.1;net461</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Markdig" Version="0.16.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' != 'netcoreapp3.1'">
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor"
Version="2.1.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.1'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

With the preceding project file:

- The `Markdig` package is added for all consumers.
- A reference to [Microsoft.AspNetCore.Mvc.Razor](#) is added for consumers targeting .NET Framework 4.6.1 or later or .NET Core 2.x. Version 2.1.0 of the package works with ASP.NET Core 2.2 because of backwards compatibility.
- The shared framework is referenced for consumers targeting .NET Core 3.x. The `Microsoft.AspNetCore.Mvc.Razor` package is included in the shared framework.

Alternatively, .NET Standard 2.0 could be targeted instead of targeting both .NET Core 2.1 and .NET Framework 4.6.1:

XML

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

  <PropertyGroup>
    <TargetFrameworks>netstandard2.0;netcoreapp3.1</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Markdig" Version="0.16.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' != 'netcoreapp3.1'">
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor"
Version="2.1.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.1'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
</Project>
```

With the preceding project file, the following caveats exist:

- Since the library only contains Tag Helpers, it's more straightforward to target the specific platforms on which ASP.NET Core runs: .NET Core and .NET Framework. Tag Helpers can't be used by other .NET Standard 2.0-compliant target frameworks such as Unity, UWP, and Xamarin.
- Using .NET Standard 2.0 from .NET Framework has some issues that were addressed in .NET Framework 4.7.2. You can improve the experience for consumers using .NET Framework 4.6.1 through 4.7.1 by targeting .NET Framework 4.6.1.

If your library needs to call platform-specific APIs, target specific .NET implementations instead of .NET Standard. For more information, see [Multi-targeting](#).

Use an API that hasn't changed

Imagine a scenario in which you're upgrading a middleware library from .NET Core 2.2 to 3.1. The ASP.NET Core middleware APIs being used in the library haven't changed between ASP.NET Core 2.2 and 3.1. To continue supporting the middleware library in .NET Core 3.1, take the following steps:

- Follow the [standard library guidance](#).
- Add a package reference for each API's NuGet package if the corresponding assembly doesn't exist in the shared framework.

Use an API that changed

Imagine a scenario in which you're upgrading a library from .NET Core 2.2 to .NET Core 3.1. An ASP.NET Core API being used in the library has a [breaking change](#) in ASP.NET Core 3.1. Consider whether the library can be rewritten to not use the broken API in all versions.

If you can rewrite the library, do so and continue to target an earlier target framework (for example, .NET Standard 2.0 or .NET Framework 4.6.1) with package references.

If you can't rewrite the library, take the following steps:

- Add a target for .NET Core 3.1.
- Add a `<FrameworkReference>` element for the shared framework.
- Use the [#if preprocessor directive](#) with the appropriate target framework symbol to conditionally compile code.

For example, synchronous reads and writes on HTTP request and response streams are disabled by default as of ASP.NET Core 3.1. ASP.NET Core 2.2 supports the synchronous behavior by default. Consider a middleware library in which synchronous reads and writes should be enabled where I/O is occurring. The library should enclose the code to enable synchronous features in the appropriate preprocessor directive. For example:

C#

```
public async Task Invoke(HttpContext httpContext)
{
    if (httpContext.Request.Path.StartsWithSegments(_path,
StringComparison.Ordinal))
    {
        httpContext.Response.StatusCode = (int) HttpStatusCode.OK;
        httpContext.Response.ContentType = "application/json";
        httpContext.Response.ContentLength = _bufferSize;
    }
}
```

```

#if !NETCOREAPP3_1 && !NETCOREAPP5_0
    var syncIOFeature =
        httpContext.Features.Get<IHttpBodyControlFeature>();
    if (syncIOFeature != null)
    {
        syncIOFeature.AllowSynchronousIO = true;
    }

    using (var sw = new StreamWriter(
        httpContext.Response.Body, _encoding, bufferSize: _bufferSize))
    {
        _json.Serialize(sw, new JsonMessage { message = "Hello, World!"
    });
    }
#else
    await JsonSerializer.SerializeAsync<JsonMessage>(
        httpContext.Response.Body, new JsonMessage { message = "Hello,
World!" });
#endif
    return;
}

await _next(httpContext);
}

```

Use an API introduced in 3.1

Imagine that you want to use an ASP.NET Core API that was introduced in ASP.NET Core 3.1. Consider the following questions:

1. Does the library functionally require the new API?
2. Can the library implement this feature in a different way?

If the library functionally requires the API and there's no way to implement it down-level:

- Target .NET Core 3.x only.
- Add a `<FrameworkReference>` element for the shared framework.

If the library can implement the feature in a different way:

- Add .NET Core 3.x as a target framework.
- Add a `<FrameworkReference>` element for the shared framework.
- Use the `#if` preprocessor directive with the appropriate target framework symbol to conditionally compile code.

For example, the following Tag Helper uses the `IWebHostEnvironment` interface introduced in ASP.NET Core 3.1. Consumers targeting .NET Core 3.1 execute the code

path defined by the `NETCOREAPP3_1` target framework symbol. The Tag Helper's constructor parameter type changes to `IHostingEnvironment` for .NET Core 2.1 and .NET Framework 4.6.1 consumers. This change was necessary because ASP.NET Core 3.1 marked `IHostingEnvironment` as obsolete and recommended `IWebHostEnvironment` as the replacement.

C#

```
[HtmlTargetElement("script", Attributes = "asp-inline")]
public class ScriptInliningTagHelper : TagHelper
{
    private readonly IFileProvider _wwwroot;

    #if NETCOREAPP3_1
        public ScriptInliningTagHelper(IWebHostEnvironment env)
    #else
        public ScriptInliningTagHelper(IHostingEnvironment env)
    #endif
    {
        _wwwroot = env.WebRootFileProvider;

        // code omitted for brevity
    }
}
```

The following multi-targeted project file supports this Tag Helper scenario:

XML

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

    <PropertyGroup>
        <TargetFrameworks>netcoreapp2.1;netcoreapp3.1;net461</TargetFrameworks>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="Markdig" Version="0.16.0" />
    </ItemGroup>

    <ItemGroup Condition="'$(TargetFramework)' != 'netcoreapp3.1'">
        <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor"
            Version="2.1.0" />
    </ItemGroup>

    <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.1'">
        <FrameworkReference Include="Microsoft.AspNetCore.App" />
    </ItemGroup>
</Project>
```

Use an API removed from the shared framework

To use an ASP.NET Core assembly that was removed from the shared framework, add the appropriate package reference. For a list of packages removed from the shared framework in ASP.NET Core 3.1, see [Remove obsolete package references](#).

For example, to add the web API client:

XML

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.WebApi.Client"
Version="5.2.7" />
  </ItemGroup>

</Project>
```

Additional resources

- [Reusable Razor UI in class libraries with ASP.NET Core](#)
- [Consume ASP.NET Core Razor components from a Razor class library \(RCL\)](#)
- [.NET implementation support](#)
- [.NET support policies](#) ↗

Microsoft.AspNetCore.App for ASP.NET Core

Article • 07/26/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

The ASP.NET Core shared framework (`Microsoft.AspNetCore.App`) contains assemblies that are developed and supported by Microsoft. `Microsoft.AspNetCore.App` is installed when the [.NET Core 3.0 or later SDK](#) is installed. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and includes a runtime component and a targeting pack. For more information, see [The shared framework](#).

- Projects that target the `Microsoft.NET.Sdk.Web` SDK implicitly reference the `Microsoft.AspNetCore.App` framework.

No additional references are required for these projects:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>
  ...
</Project>
```

The ASP.NET Core shared framework:

- Doesn't include third-party dependencies.
- Includes all supported packages by the ASP.NET Core team.

Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.0

Article • 07/26/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

Note

The `Microsoft.AspNetCore.All` metapackage isn't used in ASP.NET Core 3.0 and later. For more information, see [this GitHub issue](#).

Note

We recommend applications targeting ASP.NET Core 2.1 and later use the [Microsoft.AspNetCore.App metapackage](#) rather than this package. See [Migrating from Microsoft.AspNetCore.All to Microsoft.AspNetCore.App](#) in this article.

This feature requires ASP.NET Core 2.x targeting .NET Core 2.x.

[Microsoft.AspNetCore.All](#) is a metapackage that refers to a 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](#).

The shared framework that `Microsoft.AspNetCore.All` refers to includes:

- All supported packages by the ASP.NET Core team.
- All supported packages by the Entity Framework Core.
- Internal and 3rd-party dependencies used by ASP.NET Core and Entity Framework Core.

All the features of ASP.NET Core 2.x and Entity Framework Core 2.x are included in the `Microsoft.AspNetCore.All` package. The default project templates targeting ASP.NET

Core 2.0 use this package.

The version number of the `Microsoft.AspNetCore.All` metapackage represents the minimum ASP.NET Core version and Entity Framework Core version.

The following `.csproj` file references the `Microsoft.AspNetCore.All` metapackage for ASP.NET Core:

XML

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

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

</Project>
```

Implicit versioning

In ASP.NET Core 2.1 or later, you can specify the `Microsoft.AspNetCore.All` package reference without a version. When the version isn't specified, an implicit version is specified by the SDK (`Microsoft.NET.Sdk.Web`). We recommend relying on the implicit version specified by the SDK and not explicitly setting the version number on the package reference. If you have questions about this approach, leave a GitHub comment at the [Discussion for the Microsoft.AspNetCore.App implicit version](#).

The implicit version is set to `major.minor.0` for portable apps. The shared framework roll-forward mechanism runs the app on the latest compatible version among the installed shared frameworks. To guarantee the same version is used in development, test, and production, ensure the same version of the shared framework is installed in all environments. For self-contained apps, the implicit version number is set to the `major.minor.patch` of the shared framework bundled in the installed SDK.

Specifying a version number on the `Microsoft.AspNetCore.All` package reference does **not** guarantee that version of the shared framework is chosen. For example, suppose version "2.1.1" is specified, but "2.1.3" is installed. In that case, the app will use "2.1.3". Although not recommended, you can disable roll forward (patch and/or minor). For more information regarding dotnet host roll-forward and how to configure its behavior, see [dotnet host roll forward](#).

The project's SDK must be set to `Microsoft.NET.Sdk.Web` in the project file to use the implicit version of `Microsoft.AspNetCore.All`. When the `Microsoft.NET.Sdk` SDK is specified (`<Project Sdk="Microsoft.NET.Sdk">` at the top of the project file), the following warning is generated:

Warning NU1604: Project dependency Microsoft.AspNetCore.All does not contain an inclusive lower bound. Include a lower bound in the dependency version to ensure consistent restore results.

This is a known issue with the .NET Core 2.1 SDK and will be fixed in the .NET Core 2.2 SDK.

Migrating from Microsoft.AspNetCore.All to Microsoft.AspNetCore.App

The following packages are included in `Microsoft.AspNetCore.All` but not the `Microsoft.AspNetCore.App` package.

- `Microsoft.AspNetCore.ApplicationInsights.HostingStartup`
- `Microsoft.AspNetCore.AzureAppServices.HostingStartup`
- `Microsoft.AspNetCore.AzureAppServicesIntegration`
- `Microsoft.AspNetCore.DataProtection.AzureKeyVault`
- `Microsoft.AspNetCore.DataProtection.AzureStorage`
- `Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv`
- `Microsoft.AspNetCore.SignalR.Redis`
- `Microsoft.Data.Sqlite`
- `Microsoft.Data.Sqlite.Core`
- `Microsoft.EntityFrameworkCore.Sqlite`
- `Microsoft.EntityFrameworkCore.Sqlite.Core`
- `Microsoft.Extensions.Caching.Redis`
- `Microsoft.Extensions.Configuration.AzureKeyVault`
- `Microsoft.Extensions.Logging.AzureAppServices`
- `Microsoft.VisualStudio.Web.BrowserLink`

To move from `Microsoft.AspNetCore.All` to `Microsoft.AspNetCore.App`, if your app uses any APIs from the above packages, or packages brought in by those packages, add references to those packages in your project.

Any dependencies of the preceding packages that otherwise aren't dependencies of `Microsoft.AspNetCore.App` are not included implicitly. For example:

- `StackExchange.Redis` as a dependency of `Microsoft.Extensions.Caching.Redis`
- `Microsoft.ApplicationInsights` as a dependency of `Microsoft.AspNetCore.ApplicationInsights.HostingStartup`

Update ASP.NET Core 2.1

We recommend migrating to the `Microsoft.AspNetCore.App` metapackage for 2.1 and later. To keep using the `Microsoft.AspNetCore.All` metapackage and ensure the latest patch version is deployed:

- On development machines and build servers: Install the latest [.NET Core SDK](#) .
- On deployment servers: Install the latest [.NET Core runtime](#) . Your app will roll forward to the latest installed version on an application restart.

High-performance logging in .NET

Article • 04/11/2024

The [LoggerMessage](#) class exposes functionality to create cacheable delegates that require fewer object allocations and reduced computational overhead compared to [logger extension methods](#), such as [LogInformation](#) and [LogDebug](#). For high-performance logging scenarios, use the [LoggerMessage](#) pattern.

[LoggerMessage](#) provides the following performance advantages over logger extension methods:

- Logger extension methods require "boxing" (converting) value types, such as `int`, into `object`. The [LoggerMessage](#) pattern avoids boxing by using static [Action](#) fields and extension methods with strongly typed parameters.
- Logger extension methods must parse the message template (named format string) every time a log message is written. [LoggerMessage](#) only requires parsing a template once when the message is defined.

Important

Instead of using the [LoggerMessage class](#) to create high-performance logs, you can use the [LoggerMessage attribute](#) in .NET 6 and later versions. The `LoggerMessageAttribute` provides source-generation logging support designed to deliver a highly usable and highly performant logging solution for modern .NET applications. For more information, see [Compile-time logging source generation \(.NET Fundamentals\)](#).

The sample app demonstrates [LoggerMessage](#) features with a priority queue processing worker service. The app processes work items in priority order. As these operations occur, log messages are generated using the [LoggerMessage](#) pattern.

Tip

All of the logging example source code is available in the [Samples Browser](#) for download. For more information, see [Browse code samples: Logging in .NET](#).

Define a logger message

Use [Define\(LogLevel, EventId, String\)](#) to create an [Action](#) delegate for logging a message. [Define](#) overloads permit passing up to six type parameters to a named format string (template).

The string provided to the [Define](#) method is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend [Pascal casing](#) for placeholder names. For example, `{Item}`, `{DateTime}`.

Each log message is an [Action](#) held in a static field created by [LoggerMessage.Define](#). For example, the sample app creates a field to describe a log message for the processing of work items:

C#

```
private static readonly Action<ILogger, Exception>
    s_failedToProcessWorkItem;
```

For the [Action](#), specify:

- The log level.
- A unique event identifier ([EventId](#)) with the name of the static extension method.
- The message template (named format string).

As work items are dequeued for processing, the worker service app sets the:

- Log level to [LogLevel.Critical](#).
- Event ID to `13` with the name of the `FailedToProcessWorkItem` method.
- Message template (named format string) to a string.

C#

```
s_failedToProcessWorkItem = LoggerMessage.Define(
    LogLevel.Critical,
    new EventId(13, nameof(FailedToProcessWorkItem)),
    "Epic failure processing item!");
```

The [LoggerMessage.Define](#) method is used to configure and define an [Action](#) delegate, which represents a log message.

Structured logging stores may use the event name when it's supplied with the event ID to enrich logging. For example, [Serilog](#) [↗](#) uses the event name.

The [Action](#) is invoked through a strongly typed extension method. The `PriorityItemProcessed` method logs a message every time a work item is processed. `FailedToProcessWorkItem` is called if and when an exception occurs:

C#

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using (IDisposable? scope = logger.ProcessingWorkScope(DateTime.Now))
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                WorkItem? nextItem =
priorityQueue.ProcessNextHighestPriority();

                if (nextItem is not null)
                {
                    logger.PriorityItemProcessed(nextItem);
                }
            }
            catch (Exception ex)
            {
                logger.FailedToProcessWorkItem(ex);
            }

            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

Inspect the app's console output:

Console

```
crit: WorkerServiceOptions.Example.Worker[13]
      Epic failure processing item!
      System.Exception: Failed to verify communications.
      at
WorkerServiceOptions.Example.Worker.ExecuteAsync(CancellationToken
stoppingToken) in
      ..\Worker.cs:line 27
```

To pass parameters to a log message, define up to six types when creating the static field. The sample app logs the work item details when processing items by defining a `WorkItem` type for the [Action](#) field:

C#

```
private static readonly Action<ILogger, WorkItem, Exception>
s_processingPriorityItem;
```

The delegate's log message template receives its placeholder values from the types provided. The sample app defines a delegate for adding a work item where the item parameter is a `WorkItem`:

C#

```
s_processingPriorityItem = LoggerMessage.Define<WorkItem>(
    LogLevel.Information,
    new EventId(1, nameof(PriorityItemProcessed)),
    "Processing priority item: {Item}");
```

The static extension method for logging that a work item is being processed, `PriorityItemProcessed`, receives the work item argument value and passes it to the `Action` delegate:

C#

```
public static void PriorityItemProcessed(
    this ILogger logger, WorkItem workItem) =>
    s_processingPriorityItem(logger, workItem, default!);
```

In the worker service's `ExecuteAsync` method, `PriorityItemProcessed` is called to log the message:

C#

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using (IDisposable? scope = logger.ProcessingWorkScope(DateTime.Now))
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                WorkItem? nextItem =
                priorityQueue.ProcessNextHighestPriority();

                if (nextItem is not null)
                {
                    logger.PriorityItemProcessed(nextItem);
                }
            }
        }
    }
}
```

```

        catch (Exception ex)
        {
            logger.FailedToProcessWorkItem(ex);
        }

        await Task.Delay(1_000, stoppingToken);
    }
}
}

```

Inspect the app's console output:

Console

```

info: WorkerServiceOptions.Example.Worker[1]
      Processing priority item: Priority-Extreme (50db062a-9732-4418-936d-
      110549ad79e4): 'Verify communications'

```

Define logger message scope

The [DefineScope\(string\)](#) method creates a [Func<TResult>](#) delegate for defining a [log scope](#). [DefineScope](#) overloads permit passing up to six type parameters to a named format string (template).

As is the case with the [Define](#) method, the string provided to the [DefineScope](#) method is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend [Pascal casing](#) for placeholder names. For example, `{Item}`, `{DateTime}`.

Define a [log scope](#) to apply to a series of log messages using the [DefineScope](#) method. Enable `IncludeScopes` in the console logger section of *appsettings.json*:

JSON

```

{
  "Logging": {
    "Console": {
      "IncludeScopes": true
    },
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}

```

```
}  
}
```

To create a log scope, add a field to hold a `Func<TResult>` delegate for the scope. The sample app creates a field named `s_processingWorkScope` (*Internal/LoggerExtensions.cs*):

C#

```
private static readonly Func<ILogger, DateTime, IDisposable?>  
s_processingWorkScope;
```

Use `DefineScope` to create the delegate. Up to six types can be specified for use as template arguments when the delegate is invoked. The sample app uses a message template that includes the date time in which processing started:

C#

```
s_processingWorkScope =  
    LoggerMessage.DefineScope<DateTime>(  
        "Processing scope, started at: {DateTime}");
```

Provide a static extension method for the log message. Include any type parameters for named properties that appear in the message template. The sample app takes in a `DateTime` for a custom time stamp to log and returns `_processingWorkScope`:

C#

```
public static IDisposable? ProcessingWorkScope(  
    this ILogger logger, DateTime time) =>  
    s_processingWorkScope(logger, time);
```

The scope wraps the logging extension calls in a `using` block:

C#

```
protected override async Task ExecuteAsync(  
    CancellationToken stoppingToken)  
{  
    using (IDisposable? scope = logger.ProcessingWorkScope(DateTime.Now))  
    {  
        while (!stoppingToken.IsCancellationRequested)  
        {  
            try  
            {  
                WorkItem? nextItem =  
priorityQueue.ProcessNextHighestPriority();
```

```

        if (nextItem is not null)
        {
            logger.PriorityItemProcessed(nextItem);
        }
    }
    catch (Exception ex)
    {
        logger.FailedToProcessWorkItem(ex);
    }

    await Task.Delay(1_000, stoppingToken);
}
}
}

```

Inspect the log messages in the app's console output. The following result shows priority ordering of log messages with the log scope message included:

Console

```

info: WorkerServiceOptions.Example.Worker[1]
    => Processing scope, started at: 04/11/2024 11:27:52
    Processing priority item: Priority-Extreme (7d153ef9-8894-4282-836a-8e5e38319fb3): 'Verify communications'
info: Microsoft.Hosting.Lifetime[0]
    Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
    Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
    Content root path: D:\source\repos\dotnet-docs\docs\core\extensions\snippets\logging\worker-service-options
info: WorkerServiceOptions.Example.Worker[1]
    => Processing scope, started at: 04/11/2024 11:27:52
    Processing priority item: Priority-High (dbad6558-60cd-4eb1-8531-231e90081f62): 'Validate collection'
info: WorkerServiceOptions.Example.Worker[1]
    => Processing scope, started at: 04/11/2024 11:27:52
    Processing priority item: Priority-Medium (1eabe213-dc64-4e3a-9920-f67fe1dfb0f6): 'Propagate selections'
info: WorkerServiceOptions.Example.Worker[1]
    => Processing scope, started at: 04/11/2024 11:27:52
    Processing priority item: Priority-Medium (1142688d-d4dc-4f78-95c5-04ec01cbfac7): 'Enter pooling [contention]'
info: WorkerServiceOptions.Example.Worker[1]
    => Processing scope, started at: 04/11/2024 11:27:52
    Processing priority item: Priority-Low (e85e0c4d-0840-476e-b8b0-22505c08e913): 'Health check network'
info: WorkerServiceOptions.Example.Worker[1]
    => Processing scope, started at: 04/11/2024 11:27:52
    Processing priority item: Priority-Deferred (07571363-d559-4e72-bc33-cd8398348786): 'Ping weather service'
info: WorkerServiceOptions.Example.Worker[1]
    => Processing scope, started at: 04/11/2024 11:27:52

```

```
Processing priority item: Priority-Deferred (2bf74f2f-0198-4831-8138-03368e60bd6b): 'Set process state'  
info: Microsoft.Hosting.Lifetime[0]  
Application is shutting down...
```

Log level guarded optimizations

Another performance optimization can be made by checking the [LogLevel](#), with [ILogger.IsEnabled\(LogLevel\)](#) before an invocation to the corresponding `Log*` method.

When logging isn't configured for the given `LogLevel`, the following statements are true:

- [ILogger.Log](#) isn't called.
- An allocation of `object[]` representing the parameters is avoided.
- Value type boxing is avoided.

For more information:

- [Micro benchmarks in the .NET runtime](#) [↗](#)
- [Background and motivation for log level checks](#) [↗](#)

See also

- [Logging in .NET](#)

Develop ASP.NET Core apps using a file watcher

Article • 05/31/2024

By [Rick Anderson](#) and [Victor Hurdugaci](#)

`dotnet watch` is a tool that runs a [.NET CLI](#) command when source files change. For example, a file change can trigger compilation, test execution, or deployment.

This tutorial uses an existing web API with two endpoints: one that returns a sum and one that returns a product. The product method has a bug, which is fixed in this tutorial.

Download the [sample app](#). It consists of two projects: *WebApp* (an ASP.NET Core web API) and *WebAppTests* (unit tests for the web API).

In a command shell, navigate to the *WebApp* folder. Run the following command:

```
.NET CLI
```

```
dotnet run
```

ⓘ Note

You can use `dotnet run --project <PROJECT>` to specify a project to run. For example, running `dotnet run --project WebApp` from the root of the sample app will also run the *WebApp* project.

The console output shows messages similar to the following (indicating that the app is running and awaiting requests):

```
Console
```

```
$ dotnet run
Hosting environment: Development
Content root path: C:/Docs/aspnetcore/tutorials/dotnet-watch/sample/WebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

In a web browser, navigate to `http://localhost:<port number>/api/math/sum?a=4&b=5`. You should see the result of 9.

Navigate to the product API (`http://localhost:<port number>/api/math/product?a=4&b=5`). It returns `9`, not `20` as you'd expect. That problem is fixed later in the tutorial.

Run .NET CLI commands using `dotnet watch`

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

 Expand table

Command	Command with watch
<code>dotnet run</code>	<code>dotnet watch run</code>
<code>dotnet run -f netcoreapp3.1</code>	<code>dotnet watch run -f netcoreapp3.1</code>
<code>dotnet run -f netcoreapp3.1 -- --arg1</code>	<code>dotnet watch run -f netcoreapp3.1 -- --arg1</code>
<code>dotnet test</code>	<code>dotnet watch test</code>

Run `dotnet watch run` in the *WebApp* folder. The console output indicates `watch` has started.

Running `dotnet watch run` on a web app launches a browser that navigates to the app's URL once ready. `dotnet watch` does this by reading the app's console output and waiting for the ready message displayed by [WebHost](#).

`dotnet watch` refreshes the browser when it detects changes to watched files. To do this, the watch command injects a middleware to the app that modifies HTML responses created by the app. The middleware adds a JavaScript script block to the page that allows `dotnet watch` to instruct the browser to refresh. Currently, changes to all watched files, including static content such as `.html` and `.css` files cause the app to be rebuilt.

`dotnet watch`:

- Only watches files that impact builds by default.
- Any additionally watched files (via configuration) still results in a build taking place.

For more information on configuration, see [dotnet-watch configuration](#) in this document.

Note

You can use `dotnet watch --project <PROJECT>` to specify a project to watch. For example, running `dotnet watch --project WebApp run` from the root of the sample

app will also run and watch the *WebApp* project.

Make changes with `dotnet watch`

Make sure `dotnet watch` is running.

Fix the bug in the `Product` method of `MathController.cs` so it returns the product and not the sum:

```
C#  
  
public static int Product(int a, int b)  
{  
    return a * b;  
}
```

Save the file. The console output indicates that `dotnet watch` detected a file change and restarted the app.

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

Run tests using `dotnet watch`

1. Change the `Product` method of `MathController.cs` back to returning the sum. Save the file.
2. In a command shell, navigate to the *WebAppTests* folder.
3. Run `dotnet restore`.
4. Run `dotnet watch test`. Its output indicates that a test failed and that the watcher is awaiting file changes:

Console

```
Total tests: 2. Passed: 1. Failed: 1. Skipped: 0.  
Test Run Failed.
```

5. Fix the `Product` method code so it returns the product. Save the file.

`dotnet watch` detects the file change and reruns the tests. The console output indicates the tests passed.

Customize files list to watch

By default, `dotnet-watch` tracks all files matching the following glob patterns:

- `**/*.cs`
- `*.csproj`
- `**/*.resx`
- Content files: `wwwroot/**`, `**/*.config`, `**/*.json`

More items can be added to the watch list by editing the `.csproj` file. Items can be specified individually or by using glob patterns.

XML

```
<ItemGroup>
  <!-- extends watching group to include *.js files -->
  <Watch Include="**\*.js"
Exclude="node_modules\**\*;**\*.js.map;obj\**\*;bin\**\*" />
</ItemGroup>
```

Opt-out of files to be watched

`dotnet-watch` can be configured to ignore its default settings. To ignore specific files, add the `Watch="false"` attribute to an item's definition in the `.csproj` file:

XML

```
<ItemGroup>
  <!-- exclude Generated.cs from dotnet-watch -->
  <Compile Include="Generated.cs" Watch="false" />

  <!-- exclude Strings.resx from dotnet-watch -->
  <EmbeddedResource Include="Strings.resx" Watch="false" />

  <!-- exclude changes in this referenced project -->
  <ProjectReference Include="..\ClassLibrary1\ClassLibrary1.csproj"
Watch="false" />
</ItemGroup>
```

XML

```
<ItemGroup>
  <!-- Exclude all Content items from being watched. -->
  <Content Update="@(<Content>)" Watch="false" />
</ItemGroup>
```

Custom watch projects

`dotnet-watch` isn't restricted to C# projects. Custom watch projects can be created to handle different scenarios. Consider the following project layout:

- **test/**
 - `UnitTests/UnitTests.csproj`
 - `IntegrationTests/IntegrationTests.csproj`

If the goal is to watch both projects, create a custom project file configured to watch both projects:

XML

```
<Project>
  <ItemGroup>
    <TestProjects Include="**\*.csproj" />
    <Watch Include="**\*.cs" />
  </ItemGroup>

  <Target Name="Test">
    <MSBuild Targets="VSTest" Projects="@ (TestProjects)" />
  </Target>

  <Import Project="$(MSBuildExtensionsPath)\Microsoft.Common.targets" />
</Project>
```

To start file watching on both projects, change to the `test` folder. Execute the following command:

.NET CLI

```
dotnet watch msbuild /t:Test
```

VSTest executes when any file changes in either test project.

dotnet-watch configuration

Some configuration options can be passed to `dotnet watch` through environment variables. The available variables are:

[Expand table](#)

Setting	Description
<code>DOTNET_USE_POLLING_FILE_WATCHER</code>	If set to "1" or "true", <code>dotnet watch</code> uses a polling file watcher instead of CoreFx's <code>FileSystemWatcher</code> . Used when watching files on network shares or Docker mounted volumes.
<code>DOTNET_WATCH_SUPPRESS_MSBUILD_INCREMENTALISM</code>	By default, <code>dotnet watch</code> optimizes the build by avoiding certain operations such as running restore or re-evaluating the set of watched files on every file change. If set to "1" or "true", these optimizations are disabled.
<code>DOTNET_WATCH_SUPPRESS_LAUNCH_BROWSER</code>	<code>dotnet watch run</code> attempts to launch browsers for web apps with <code>launchBrowser</code> configured in <code>launchSettings.json</code> . If set to "1" or "true", this behavior is suppressed.
<code>DOTNET_WATCH_SUPPRESS_BROWSER_REFRESH</code>	<code>dotnet watch run</code> attempts to refresh browsers when it detects file changes. If set to "1" or "true", this behavior is suppressed. This behavior is also suppressed if <code>DOTNET_WATCH_SUPPRESS_LAUNCH_BROWSER</code> is set.

Browser refresh

`dotnet watch` injects a script into the app that allows it to refresh the browser when the content changes. In some scenarios, such as when the app enables response compression, `dotnet watch` might *not* be able to inject the script. For such cases in development, manually inject the script into the app. For example, to configure the web app to manually inject the script, update the layout file to include `_framework/aspnet-browser-refresh.js`:

```
razor
```

```
@* _Layout.cshtml *@
<environment names="Development">
  <script src="/_framework/aspnetcore-browser-refresh.js"></script>
</environment>
```

Non-ASCII characters

Visual Studio 17.2 and later includes the .NET SDK 6.0.300 and later. With the .NET SDK and 6.0.300 later, `dotnet-watch` emits non-ASCII characters to the console during a hot

reload session. On certain console hosts, such as the Windows conhost, these characters may appear garbled. To avoid garbled characters, consider one of the following approaches:

- Configure the `DOTNET_WATCH_SUPPRESS_EMOJIS=1` environment variable to suppress emitting these values.
- Switch to a different terminal, such as <https://github.com/microsoft/terminal> [↗], that supports rendering non-ASCII characters.

Factory-based middleware activation in ASP.NET Core

Article • 07/26/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

[IMiddlewareFactory/IMiddleware](#) is an extensibility point for [middleware](#) activation that offers the following benefits:

- Activation per client request (injection of scoped services)
- Strong typing of middleware

[UseMiddleware](#) extension methods check if a middleware's registered type implements [IMiddleware](#). If it does, the [IMiddlewareFactory](#) instance registered in the container is used to resolve the [IMiddleware](#) implementation instead of using the convention-based middleware activation logic. The middleware is registered as a [scoped or transient service](#) in the app's service container.

[IMiddleware](#) is activated per client request (connection), so scoped services can be injected into the middleware's constructor.

IMiddleware

[IMiddleware](#) defines middleware for the app's request pipeline. The [InvokeAsync\(HttpContext, RequestDelegate\)](#) method handles requests and returns a [Task](#) that represents the execution of the middleware.

Middleware activated by convention:

C#

```
public class ConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public ConventionalMiddleware(RequestDelegate next)
```

```

=> _next = next;

    public async Task InvokeAsync(HttpContext context, SampleDbContext
dbContext)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrEmpty(keyValue))
        {
            dbContext.Requests.Add(new Request("Conventional", keyValue));

            await dbContext.SaveChangesAsync();
        }

        await _next(context);
    }
}

```

Middleware activated by [MiddlewareFactory](#):

C#

```

public class FactoryActivatedMiddleware : IMiddleware
{
    private readonly SampleDbContext _dbContext;

    public FactoryActivatedMiddleware(SampleDbContext dbContext)
        => _dbContext = dbContext;

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrEmpty(keyValue))
        {
            _dbContext.Requests.Add(new Request("Factory", keyValue));

            await _dbContext.SaveChangesAsync();
        }

        await next(context);
    }
}

```

Extensions are created for the middleware:

C#

```

public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseConventionalMiddleware(
        this IApplicationBuilder app)
    {
    }
}

```

```

=> app.UseMiddleware<ConventionalMiddleware>();

public static IApplicationBuilder UseFactoryActivatedMiddleware(
    this IApplicationBuilder app)
=> app.UseMiddleware<FactoryActivatedMiddleware>();
}

```

It isn't possible to pass objects to the factory-activated middleware with `UseMiddleware`:

```

C#

public static IApplicationBuilder UseFactoryActivatedMiddleware(
    this IApplicationBuilder app, bool option)
{
    // Passing 'option' as an argument throws a NotSupportedException at
    // runtime.
    return app.UseMiddleware<FactoryActivatedMiddleware>(option);
}

```

The factory-activated middleware is added to the built-in container in `Program.cs`:

```

C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<SampleDbContext>
    (options => options.UseInMemoryDatabase("SampleDb"));

builder.Services.AddTransient<FactoryActivatedMiddleware>();

```

Both middleware are registered in the request processing pipeline, also in `Program.cs`:

```

C#

var app = builder.Build();

app.UseConventionalMiddleware();
app.UseFactoryActivatedMiddleware();

```

IMiddlewareFactory

`IMiddlewareFactory` provides methods to create middleware. The middleware factory implementation is registered in the container as a scoped service.

The default `IMiddlewareFactory` implementation, `MiddlewareFactory`, is found in the `Microsoft.AspNetCore.Http` package.

Additional resources

- [View or download sample code](#) [↗] (how to download)
- [ASP.NET Core Middleware](#)
- [Middleware activation with a third-party container in ASP.NET Core](#)

Middleware activation with a third-party container in ASP.NET Core

Article • 07/26/2024


Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).




This article demonstrates how to use `IMiddlewareFactory` and `IMiddleware` as an extensibility point for `middleware` activation with a third-party container. For introductory information on `IMiddlewareFactory` and `IMiddleware`, see [Factory-based middleware activation in ASP.NET Core](#).

[View or download sample code](#)  ([how to download](#))

The sample app demonstrates middleware activation by an `IMiddlewareFactory` implementation, `SimpleInjectorMiddlewareFactory`. The sample uses the [Simple Injector](#)  dependency injection (DI) container.

The sample's middleware implementation records the value provided by a query string parameter (`key`). The middleware uses an injected database context (a scoped service) to record the query string value in an in-memory database.

Note

The sample app uses [Simple Injector](#)  purely for demonstration purposes. Use of Simple Injector isn't an endorsement. Middleware activation approaches described in the Simple Injector documentation and GitHub issues are recommended by the maintainers of Simple Injector. For more information, see the [Simple Injector documentation](#)  and [Simple Injector GitHub repository](#) .

IMiddlewareFactory

`IMiddlewareFactory` provides methods to create middleware.

In the sample app, a middleware factory is implemented to create a `SimpleInjectorActivatedMiddleware` instance. The middleware factory uses the Simple Injector container to resolve the middleware:

C#

```
public class SimpleInjectorMiddlewareFactory : IMiddlewareFactory
{
    private readonly Container _container;

    public SimpleInjectorMiddlewareFactory(Container container)
    {
        _container = container;
    }

    public IMiddleware Create(Type middlewareType)
    {
        return _container.GetInstance(middlewareType) as IMiddleware;
    }

    public void Release(IMiddleware middleware)
    {
        // The container is responsible for releasing resources.
    }
}
```

IMiddleware

`IMiddleware` defines middleware for the app's request pipeline.

Middleware activated by an `IMiddlewareFactory` implementation
(Middleware/SimpleInjectorActivatedMiddleware.cs):

C#

```
public class SimpleInjectorActivatedMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public SimpleInjectorActivatedMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrEmpty(keyValue))
        {
            // ...
        }
    }
}
```

```

    {
        _db.Add(new Request()
        {
            DT = DateTime.UtcNow,
            MiddlewareActivation =
"SimpleInjectorActivatedMiddleware",
            Value = keyValue
        });

        await _db.SaveChangesAsync();
    }

    await next(context);
}
}

```

An extension is created for the middleware (`Middleware/MiddlewareExtensions.cs`):

```

C#

public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSimpleInjectorActivatedMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<SimpleInjectorActivatedMiddleware>();
    }
}

```

`Startup.ConfigureServices` must perform several tasks:

- Set up the Simple Injector container.
- Register the factory and middleware.
- Make the app's database context available from the Simple Injector container.

```

C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    // Replace the default middleware factory with the
    // SimpleInjectorMiddlewareFactory.
    services.AddTransient<IMiddlewareFactory>(_ =>
    {
        return new SimpleInjectorMiddlewareFactory(_container);
    });

    // Wrap ASP.NET Core requests in a Simple Injector execution
    // context.
    services.UseSimpleInjectorAspNetRequestScoping(_container);
}

```

```

// Provide the database context from the Simple
// Injector container whenever it's requested from
// the default service container.
services.AddScoped<AppDbContext>(provider =>
    _container.GetInstance<AppDbContext>());

_container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

_container.Register<AppDbContext>(() =>
{
    var optionsBuilder = new DbContextOptionsBuilder<DbContext>();
    optionsBuilder.UseInMemoryDatabase("InMemoryDb");
    return new AppDbContext(optionsBuilder.Options);
}, Lifestyle.Scoped);

_container.Register<SimpleInjectorActivatedMiddleware>();

_container.Verify();
}

```

The middleware is registered in the request processing pipeline in `Startup.Configure`:

C#

```

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

    app.UseSimpleInjectorActivatedMiddleware();

    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}

```

Additional resources

- [Middleware](#)

- [Factory-based middleware activation](#)
- [Simple Injector GitHub repository](#) ↗
- [Simple Injector documentation](#) ↗

Migrate from ASP.NET Core in .NET 8 to ASP.NET Core in .NET 9

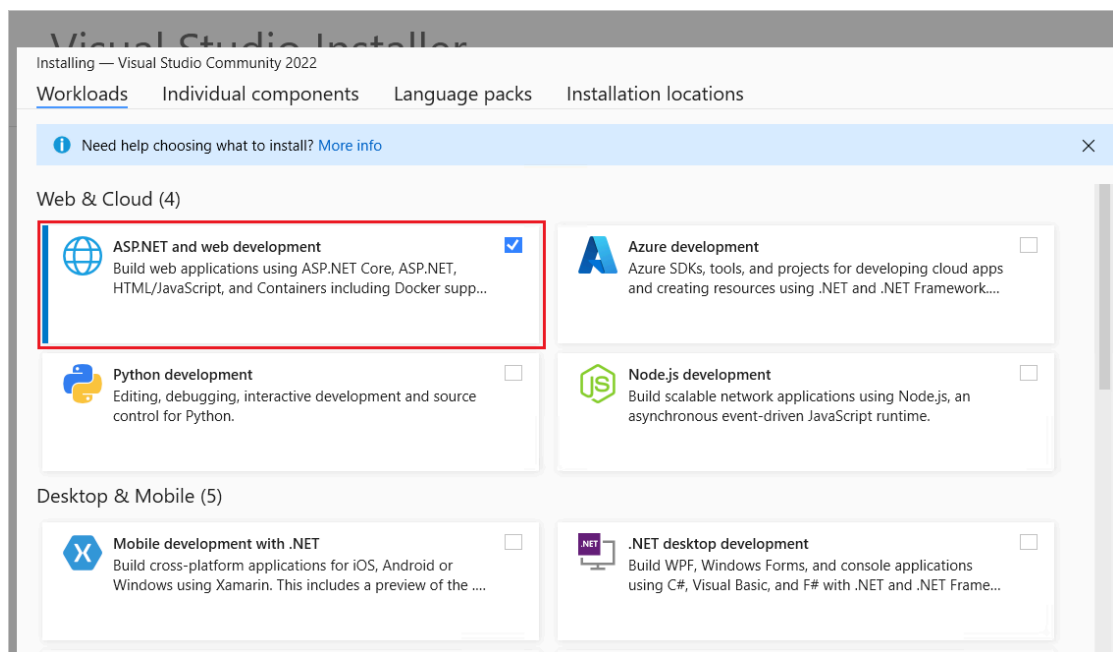
Article • 11/06/2024

This article explains how to update an ASP.NET Core in .NET 8 to ASP.NET Core in .NET 9.

Prerequisites

Visual Studio

- [Visual Studio 2022 Preview](#) with the **ASP.NET and web development** workload.



Update the .NET SDK version in `global.json`

If you rely on a `global.json` file to target a specific .NET Core SDK version, update the `version` property to the .NET 9.0 SDK version that's installed. For example:

```
diff
```

```
{
  "sdk": {
    - "version": "8.0.100"
```

```
+   "version": "9.0.100"  
  }  
}
```

Update the target framework

Update the project file's [Target Framework Moniker \(TFM\)](#) to `net9.0`:

```
diff  
  
<Project Sdk="Microsoft.NET.Sdk.Web">  
  
  <PropertyGroup>  
-    <TargetFramework>net8.0</TargetFramework>  
+    <TargetFramework>net9.0</TargetFramework>  
  </PropertyGroup>  
  
</Project>
```

Update package references

In the project file, update each [Microsoft.AspNetCore.*](#), [Microsoft.EntityFrameworkCore.*](#), [Microsoft.Extensions.*](#), and [System.Net.Http.Json](#) package reference's `Version` attribute to 9.0.0 or later. For example:

```
diff  
  
<ItemGroup>  
-  <PackageReference Include="Microsoft.AspNetCore.JsonPatch"  
Version="8.0.2" />  
-  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"  
Version="8.0.2" />  
-  <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"  
Version="8.0.0" />  
-  <PackageReference Include="System.Net.Http.Json" Version="8.0.0" />  
+  <PackageReference Include="Microsoft.AspNetCore.JsonPatch"  
Version="9.0.0" />  
+  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"  
Version="9.0.0" />  
+  <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"  
Version="9.0.0" />  
+  <PackageReference Include="System.Net.Http.Json" Version="9.0.0" />  
</ItemGroup>
```

Replace `UseStaticFiles` with `MapStaticAssets`

Optimize the handling of static files in your web apps by replacing [UseStaticFiles](#) with [MapStaticAssets](#) in the app's `Program` file:

diff

```
- app.UseStaticFiles();  
+ app.MapStaticAssets();
```

In MVC & Razor Pages apps you additionally need to chain a call to `.WithStaticAssets` after `MapRazorPages` or `MapControllerRoute` in `Program.cs`. For an example, see the [Static files in ASP.NET Core](#).

ASP.NET Core automatically fingerprints and precompresses your static files at build and publish time, and then [MapStaticAssets](#) surfaces the optimized files as endpoints using endpoint routing with appropriate caching headers.

To resolve the fingerprinted file names from your app:

- In Blazor apps, use the [ComponentBase.Assets](#) property. Update explicit references to static assets in Razor component files (`.razor`) to use `@Assets["{ASSET PATH}"]`, where the `{ASSET PATH}` placeholder is the path to the asset. Note that this should **NOT** be done for the Blazor framework scripts (`blazor.*.js`). In the following example, Bootstrap, the Blazor project template app stylesheet (`app.css`), and the [CSS isolation stylesheet](#) (based on an app's namespace of `BlazorSample`) are linked in a root component, typically the `App` component (`Components/App.razor`):

razor

```
<link rel="stylesheet" href="@Assets["bootstrap/bootstrap.min.css"]" />  
<link rel="stylesheet" href="@Assets["app.css"]" />  
<link rel="stylesheet" href="@Assets["BlazorSample.styles.css"]" />
```

- In MVC & Razor Pages apps, the script and link tag helpers will automatically resolve the fingerprinted file names.

To resolve the fingerprinted file names when importing JavaScript modules, add a generated [import map](#) [↗](#):

- In Blazor apps, add the ([ImportMap](#)) component to the `<head>` content of the app's root component, typically in the `App` component (`App.razor`):

razor


```
<ImportMap />
```

- In MVC & Razor pages apps, add `<script type="importmap"></script>` to the head of the main layout file, which is updated by the Import Map Tag Helper.

For more information, see the following resources:

- [What's new in ASP.NET Core 9.0](#)
- [ASP.NET Core Blazor static files](#)

Blazor

Adopt simplified authentication state serialization for Blazor Web Apps

Blazor Web Apps can optionally adopt [simplified authentication state serialization](#).

In the server project:

- Remove the Persisting Authentication State Provider (`PersistingAuthenticationStateProvider.cs`).
- Remove the service registration from the `Program` file. Instead, chain a call to [AddAuthenticationStateSerialization](#) on [AddRazorComponents](#):

diff

```
- builder.Services.AddScoped<AuthenticationStateProvider,  
PersistingAuthenticationStateProvider>();  
  
builder.Services.AddRazorComponents()  
    .AddInteractiveServerComponents()  
    .AddInteractiveWebAssemblyComponents()  
+    .AddAuthenticationStateSerialization();
```

The API only serializes the server-side name and role claims for access in the browser. To include all claims, set [SerializeAllClaims](#) to `true`:

C#

```
.AddAuthenticationStateSerialization(options => options.SerializeAllClaims =  
true);
```

In the client project (`.Client`):

- Remove the Persistent Authentication State Provider (`PersistentAuthenticationStateProvider.cs`).
- Remove the service registration from the `Program` file. Instead, call [AddAuthenticationStateDeserialization](#) on the service collection:

diff

```
- builder.Services.AddSingleton<AuthenticationStateProvider,  
PersistentAuthenticationStateProvider>();  
+ builder.Services.AddAuthenticationStateDeserialization();
```

For more information, see [What's new in ASP.NET Core 9.0](#).

Additional resources

Migrate from ASP.NET Core in .NET 7 to .NET 8

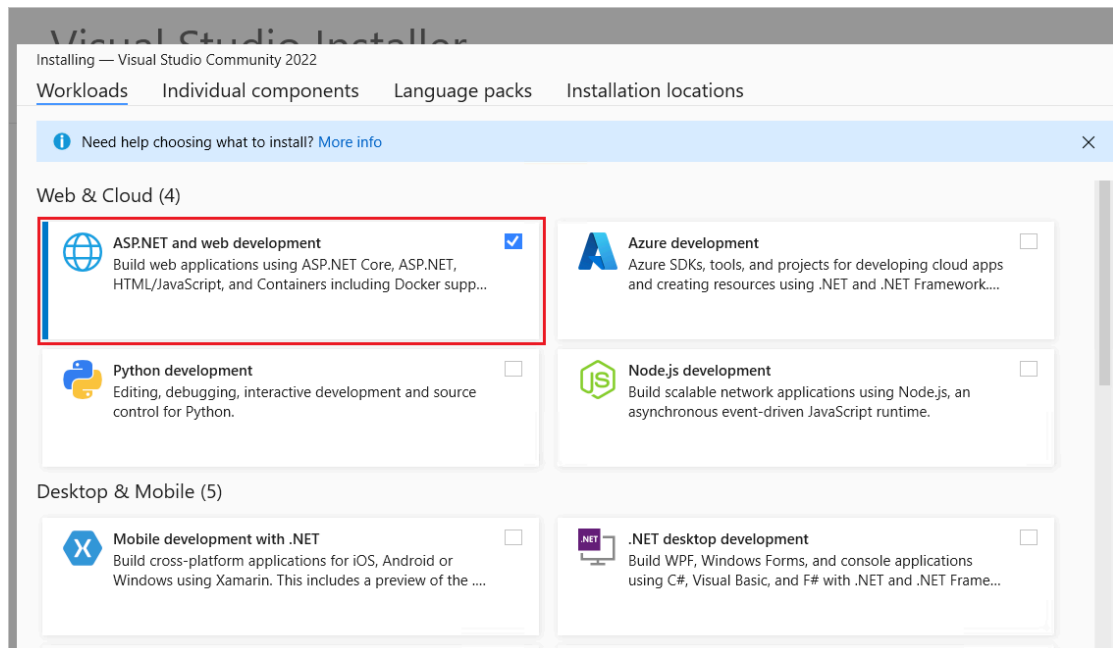
Article • 10/25/2024

This article explains how to update an existing ASP.NET Core 7.0 project to ASP.NET Core 8.0.

Prerequisites

Visual Studio

- [Visual Studio 2022](#) with the **ASP.NET and web development** workload.



Update the .NET SDK version in `global.json`

If you rely on a `global.json` file to target a specific .NET Core SDK version, update the `version` property to the .NET 8.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
    - "version": "7.0.100"
    + "version": "8.0.100"
```

```
}  
}
```

Update the target framework

Update the project file's [Target Framework Moniker \(TFM\)](#) to `net8.0`:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Web">  
  
  <PropertyGroup>  
-    <TargetFramework>net7.0</TargetFramework>  
+    <TargetFramework>net8.0</TargetFramework>  
  </PropertyGroup>  
  
</Project>
```

Update package references

In the project file, update each [Microsoft.AspNetCore.*](#), [Microsoft.EntityFrameworkCore.*](#), [Microsoft.Extensions.*](#), and [System.Net.Http.Json](#) package reference's `Version` attribute to 8.00 or later. For example:

diff

```
<ItemGroup>  
-  <PackageReference Include="Microsoft.AspNetCore.JsonPatch"  
Version="7.0.12" />  
-  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"  
Version="7.0.12" />  
-  <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"  
Version="7.0.0" />  
-  <PackageReference Include="System.Net.Http.Json" Version="7.0.1" />  
+  <PackageReference Include="Microsoft.AspNetCore.JsonPatch"  
Version="8.0.0" />  
+  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"  
Version="8.0.0" />  
+  <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"  
Version="8.0.0" />  
+  <PackageReference Include="System.Net.Http.Json" Version="8.0.0" />  
</ItemGroup>
```

Blazor

The following migration scenarios are covered:

- [Update a Blazor Server app](#)
- [Adopt all Blazor Web App conventions](#)
- [Convert a Blazor Server app into a Blazor Web App](#)
- [Update a Blazor WebAssembly app](#)
- [Convert a hosted Blazor WebAssembly app into a Blazor Web App](#)
- [Update service and endpoint option configuration](#)
- [Drop Blazor Server with Yarp routing workaround](#)
- [Migrate CascadingValue components in layout components](#)
- [Migrate the BlazorEnableCompression MSBuild property](#)
- [Migrate the <CascadingAuthenticationState> component to cascading authentication state services](#)
- *New article:* [HTTP caching issues during migration](#)
- *New article:* [New article on class libraries with static server-side rendering \(static SSR\)](#)
- [Discover components from additional assemblies](#)
- [Drop \[Parameter\] attribute when the parameter is supplied from a query string](#)
- [Blazor Server script fallback policy authorization](#)

For guidance on adding Blazor support to an ASP.NET Core app, see [Integrate ASP.NET Core Razor components into ASP.NET Core apps](#).

Update a Blazor Server app

We recommend using Blazor Web Apps in .NET 8, but Blazor Server is supported. To continue using Blazor Server with .NET 8, follow the guidance in the first three sections of this article:

- [Update the .NET SDK version in global.json](#)
- [Update the target framework](#)
- [Update package references](#)

New Blazor features introduced for Blazor Web Apps aren't available to a Blazor Server app updated to run under .NET 8. If you wish to adopt the new .NET 8 Blazor features, follow the guidance in either of the following sections:

- [Adopt all Blazor Web App conventions](#)
- [Convert a Blazor Server app into a Blazor Web App](#)

Adopt all Blazor Web App conventions

To optionally adopt all of the new Blazor Web App conventions, we recommend the following process:

- Create a new app from the Blazor Web App project template. For more information, see [Tooling for ASP.NET Core Blazor](#).
- Move the your app's components and code to the new Blazor Web App, making modifications to adopt new features.
- Update the layout and styles of the Blazor Web App.

New .NET 8 features are covered in [What's new in ASP.NET Core 8.0](#). When updating an app from .NET 6 or earlier, see the migration and release notes (*What's new* articles) for intervening releases.

Convert a Blazor Server app into a Blazor Web App

Blazor Server apps are supported in .NET 8 without any code changes. Use the following guidance to convert a Blazor Server app into an equivalent .NET 8 Blazor Web App, which makes all of the [new .NET 8 features](#) available.

Important

This section focuses on the minimal changes required to convert a .NET 7 Blazor Server app into a .NET 8 Blazor Web App. To adopt all of the new Blazor Web App conventions, follow the guidance in the [Adopt all Blazor Web App conventions](#) section.

1. Follow the guidance in the first three sections of this article:
 - [Update the .NET SDK version in global.json](#)
 - [Update the target framework](#)
 - [Update package references](#)
2. Move the contents of the `App` component (`App.razor`) to a new `Routes` component file (`Routes.razor`) added to the project's root folder. Leave the empty `App.razor` file in the app in the project's root folder.
3. Add an entry to the `_Imports.razor` file to make shorthand render modes available to the app:

```
razor
```

```
@using static Microsoft.AspNetCore.Components.Web.RenderMode
```

4. Move the content in the `_Host` page (`Pages/_Host.cshtml`) to the empty `App.razor` file. Proceed to make the following changes to the `App` component.

ⓘ **Note**

In the following example, the project's namespace is `BlazorServerApp`. Adjust the namespace to match your project.

Remove the following lines from the top of the file:

diff

```
- @page "/"
- @using Microsoft.AspNetCore.Components.Web
- @namespace BlazorServerApp.Pages
- @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Replace the preceding lines with a line that injects an `IHostEnvironment` instance:

razor

```
@inject IHostEnvironment Env
```

Remove the tilde (~) from the `href` of the `<base>` tag and replace with the base path for your app:

diff

```
- <base href="~/ " />
+ <base href="/" />
```

Remove the Component Tag Helper for the `HeadOutlet` component and replace it with the `HeadOutlet` component.

Remove the following line:

diff

```
- <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />
```

Replace the preceding line with the following:

razor

```
<HeadOutlet @rendermode="InteractiveServer" />
```

Remove the Component Tag Helper for the `App` component and replace it with the `Routes` component.

Remove the following line:

diff

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

Replace the preceding line with the following:

razor

```
<Routes @rendermode="InteractiveServer" />
```

ⓘ Note

The preceding configuration assumes that the app's components adopt interactive server rendering. For more information, including how to adopt static server-side rendering (SSR), see [ASP.NET Core Blazor render modes](#).

Remove the Environment Tag Helpers for error UI and replace them with the following Razor markup.

Remove the following lines:

diff

```
- <environment include="Staging,Production">
-     An error has occurred. This application may no longer respond
until reloaded.
- </environment>
- <environment include="Development">
-     An unhandled exception has occurred. See browser dev tools for
details.
- </environment>
```

Replace the preceding lines with the following:

razor


```
@if (Env.IsDevelopment())
{
    <text>
        An unhandled exception has occurred. See browser dev tools for
        details.
    </text>
}
else
{
    <text>
        An error has occurred. This app may no longer respond until
        reloaded.
    </text>
}
```

Change the Blazor script from `blazor.server.js` to `blazor.web.js`:

diff

```
- <script src="_framework/blazor.server.js"></script>
+ <script src="_framework/blazor.web.js"></script>
```

5. Delete the `Pages/_Host.cshtml` file.

6. Update `Program.cs`:

ⓘ Note

In the following example, the project's namespace is `BlazorServerApp`. Adjust the namespace to match your project.

Add a `using` statement to the top of the file for the project's namespace:

C#

```
using BlazorServerApp;
```

Replace `AddServerSideBlazor` with `AddRazorComponents` and a chained call to `AddInteractiveServerComponents`.

Remove the following line:

diff

```
- builder.Services.AddServerSideBlazor();
```

Replace the preceding line with Razor component and interactive server component services. Calling [AddRazorComponents](#) adds antiforgery services ([AddAntiforgery](#)) by default.

```
C#

builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
```

Remove the following line:

```
diff

- app.MapBlazorHub();
```

Replace the preceding line with a call to [MapRazorComponents](#), supplying the `App` component as the root component type, and add a chained call to [AddInteractiveServerRenderMode](#):

```
C#

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

Remove the following line:

```
diff

- app.MapFallbackToPage("/_Host");
```

Remove Routing Middleware:

```
diff

- app.UseRouting();
```

Add [Antiforgery Middleware](#) to the request processing pipeline after the line that adds HTTPS Redirection Middleware (`app.UseHttpsRedirection`):

```
C#
```

```
app.UseAntiforgery();
```

The preceding call to `app.UseAntiforgery` must be placed after calls, if present, to `app.UseAuthentication` and `app.UseAuthorization`. There's no need to explicitly add antiforgery services (`builder.Services.AddAntiforgery`), as they're added automatically by [AddRazorComponents](#), which was covered earlier.

7. If the Blazor Server app was configured to disable prerendering, you can continue to disable prerendering for the updated app. In the `App` component, change the value assigned to the `@rendermode` Razor directive attributes for the `HeadOutlet` and `Routes` components.

Change the value of the `@rendermode` directive attribute for both the `HeadOutlet` and `Routes` components to disable prerendering:

```
diff
```

```
- @rendermode="InteractiveServer"  
+ @rendermode="new InteractiveServerRenderMode(prerender: false)"
```

For more information, see [ASP.NET Core Blazor render modes](#).

Update a Blazor WebAssembly app

Follow the guidance in the first three sections of this article:

- [Update the .NET SDK version in global.json](#)
- [Update the target framework](#)
- [Update package references](#)

For apps that adopt [lazy assembly loading](#), change the file extension from `.dll` to `.wasm` in the app's implementation to reflect Blazor WebAssembly's adoption of [Webcil assembly packaging](#).

Prior to the release of .NET 8, guidance in [Deployment layout for ASP.NET Core hosted Blazor WebAssembly apps](#) addresses environments that block clients from downloading and executing DLLs with a multipart bundling approach. In .NET 8 or later, Blazor uses the Webcil file format to address this problem. Multipart bundling using the experimental NuGet package described by the *WebAssembly deployment layout* article isn't supported for Blazor apps in .NET 8 or later. If you desire to continue using the multipart bundle package in .NET 8 or later apps, you can use the guidance in the article

to create your own multipart bundling NuGet package, but it won't be supported by Microsoft.

Convert a hosted Blazor WebAssembly app into a Blazor Web App

Blazor WebAssembly apps are supported in .NET 8 without any code changes. Use the following guidance to convert an ASP.NET Core hosted Blazor WebAssembly app into an equivalent .NET 8 Blazor Web App, which makes all of the [new .NET 8 features](#) available.

Important

This section focuses on the minimal changes required to convert a .NET 7 ASP.NET Core hosted Blazor WebAssembly app into a .NET 8 Blazor Web App. To adopt all of the new Blazor Web App conventions, follow the guidance in the [Adopt all Blazor Web App conventions](#) section.

1. Follow the guidance in the first three sections of this article:

- [Update the .NET SDK version in global.json](#)
- [Update the target framework](#)
- [Update package references](#)

Important

Using the preceding guidance, update the `.Client`, `.Server`, and `.Shared` projects of the solution.

2. In the `.Client` project file (`.csproj`), add the following MSBuild properties:

XML

```
<NoDefaultLaunchSettingsFile>true</NoDefaultLaunchSettingsFile>
<StaticWebAssetProjectMode>Default</StaticWebAssetProjectMode>
```

Also in the `.Client` project file, remove the

[Microsoft.AspNetCore.Components.WebAssembly.DevServer](#)  package reference:

diff

```
- <PackageReference
```

```
Include="Microsoft.AspNetCore.Components.WebAssembly.DevServer"... />
```

3. Move the file content from the `.Client/wwwroot/index.html` file to a new `App` component file (`App.razor`) created at the root of the `.Server` project. After you move the file's contents, delete the `index.html` file.

Rename `App.razor` in the `.Client` project to `Routes.razor`.

In `Routes.razor`, update the value of the `AppAssembly` attribute to `typeof(Program).Assembly`.

4. In the `.Client` project, add an entry to the `_Imports.razor` file to make shorthand render modes available to the app:

```
razor
```

```
@using static Microsoft.AspNetCore.Components.Web.RenderMode
```

Make a copy of the `.Client` project's `_Imports.razor` file and add it to the `.Server` project.

5. Make the following changes to the `App.razor` file:

Replace the website's default website title (`<title>...</title>`) with a [HeadOutlet](#) component. Note the website title for use later and remove the title tags and title:

```
diff
```

```
- <title>...</title>
```

Where you removed the title, place a [HeadOutlet](#) component assigning the Interactive WebAssembly render mode (prerendering disabled):

```
razor
```

```
<HeadOutlet @rendermode="new  
InteractiveWebAssemblyRenderMode(prerender: false)" />
```

Change the [CSS style bundle](#):

```
diff
```

```
- <link href="{CLIENT PROJECT ASSEMBLY NAME}.styles.css"  
rel="stylesheet">
```

```
+ <link href="{SERVER PROJECT ASSEMBLY NAME}.styles.css"
rel="stylesheet">
```

Placeholders in the preceding code:

- {CLIENT PROJECT ASSEMBLY NAME}: Client project assembly name. Example:
BlazorSample.Client
- {SERVER PROJECT ASSEMBLY NAME}: Server project assembly name. Example:
BlazorSample.Server

Locate following `<div>...</div>` HTML markup:

diff

```
- <div id="app">
-   ...
- </div>
```

Replace the preceding `<div>...</div>` HTML markup with the `Routes` component using the Interactive WebAssembly render mode (prerendering disabled):

razor

```
<Routes @rendermode="new InteractiveWebAssemblyRenderMode(prerender:
false)" />
```

Update the `blazor.webassembly.js` script to `blazor.web.js`:

diff

```
- <script src="_framework/blazor.webassembly.js"></script>
+ <script src="_framework/blazor.web.js"></script>
```

6. Open the `.Client` project's layout file (`.Client/Shared/MainLayout.razor`) and add a `PageTitle` component with the website's default title (`{TITLE}` placeholder):

razor

```
<PageTitle>{TITLE}</PageTitle>
```

❗ Note

Other layout files should also receive a [PageTitle](#) component with the default website title.

For more information, see [Control head content in ASP.NET Core Blazor apps](#).

7. Remove the following lines from `.Client/Program.cs`:

diff

```
- builder.RootComponents.Add<App>("#app");  
- builder.RootComponents.Add<HeadOutlet>("head::after");
```

8. Update `.Server/Program.cs`:

Add Razor component and interactive WebAssembly component services to the project. Call [AddRazorComponents](#) with a chained call to [AddInteractiveWebAssemblyComponents](#). Calling [AddRazorComponents](#) adds antiforgery services ([AddAntiforgery](#)) by default.

C#

```
builder.Services.AddRazorComponents()  
    .AddInteractiveWebAssemblyComponents();
```

Add [Antiforgery Middleware](#) to the request processing pipeline.

Place the following line after the call to `app.UseHttpsRedirection`. The call to `app.UseAntiforgery` must be placed after calls, if present, to `app.UseAuthentication` and `app.UseAuthorization`. There's no need to explicitly add antiforgery services (`builder.Services.AddAntiforgery`), as they're added automatically by [AddRazorComponents](#), which was covered earlier.

C#

```
app.UseAntiforgery();
```

Remove the following line:

diff

```
- app.UseBlazorFrameworkFiles();
```

Remove the following line:

```
diff
- app.MapFallbackToFile("index.html");
```

Replace the preceding line with a call to [MapRazorComponents](#), supplying the `App` component as the root component type, and add chained calls to [AddInteractiveWebAssemblyRenderMode](#) and [AddAdditionalAssemblies](#):

```
C#
app.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode()
    .AddAdditionalAssemblies(typeof({CLIENT APP
NAMESPACE})._Imports).Assembly);
```

In the preceding example, the `{CLIENT APP NAMESPACE}` placeholder is the namespace of the `.Client` project (for example, `HostedBlazorApp.Client`).

9. Run the solution from the `.Server` project:

For Visual Studio, confirm that the `.Server` project is selected in **Solution Explorer** when running the app.

If using the .NET CLI, run the project from the `.Server` project's folder.

Update service and endpoint option configuration

With the release of Blazor Web Apps in .NET 8, Blazor service and endpoint option configuration is updated with the introduction of new API for interactive component services and component endpoint configuration.

Updated configuration guidance appears in the following locations:

- [Setting and reading the app's environment](#): Contains updated guidance, especially in the section titled *Read the environment client-side in a Blazor Web App*.
- [Server-side circuit handler options](#): Covers new Blazor-SignalR circuit and hub options configuration.
- [Render Razor components from JavaScript](#): Covers dynamic component registration with [RegisterForJavaScript](#).
- [Blazor custom elements: Blazor Web App registration](#): Covers root component custom element registration with `RegisterCustomElement`.

- [Prefix for Blazor WebAssembly assets](#): Covers control of the path string that indicates the prefix for Blazor WebAssembly assets.
- [Temporary redirection URL validity duration](#): Covers control of the lifetime of data protection validity for temporary redirection URLs emitted by Blazor server-side rendering.
- [Detailed errors](#): Covers enabling detailed errors for Razor component server-side rendering.
- [Prerendering configuration](#): Prerendering is enabled by default for Blazor Web Apps. Follow this link for guidance on how to disable prerendering if you have special circumstances that require an app to disable prerendering.
- [Form binding options](#): Covers form binding options configuration.

Drop Blazor Server with Yarp routing workaround

If you previously followed the guidance in [Enable ASP.NET Core Blazor Server support with Yarp in incremental migration](#) for migrating a Blazor Server app with Yarp to .NET 6 or .NET 7, you can reverse the workaround steps that you took when following the article's guidance. Routing and deep linking for Blazor Server with Yarp work correctly in .NET 8.

Migrate `CascadingValue` components in layout components

Cascading parameters don't pass data across render mode boundaries, and layouts are statically rendered in otherwise interactive apps. Therefore, apps that seek to use cascading parameters in interactively rendered components won't be able to cascade the values from a layout.

The two approaches for migration are:

- *(Recommended)* Pass the state as a root-level cascading value. For more information, see [Root-level cascading values](#).
- Wrap the router in the `Routes` component with the `CascadingValue` component and make the `Routes` component interactively rendered. For an example, see [CascadingValue component](#).

For more information, see [Cascading values/parameters and render mode boundaries](#).

Migrate the `BlazorEnableCompression` MSBuild property

For Blazor WebAssembly apps that disable compression and target .NET 7 or earlier but are built with the .NET 8 SDK, the `BlazorEnableCompression` MSBuild property has changed to `CompressionEnabled`:

diff

```
<PropertyGroup>
-   <BlazorEnableCompression>false</BlazorEnableCompression>
+   <CompressionEnabled>false</CompressionEnabled>
</PropertyGroup>
```

When using the .NET CLI publish command, use the new property:

.NET CLI

```
dotnet publish -p:CompressionEnabled=false
```

For more information, see the following resources:

- [Static Web Assets Compression Flag Breaking Change \(dotnet/announcements #283\)](#) [↗](#)
- [Host and deploy ASP.NET Core Blazor WebAssembly](#)

Migrate the `<CascadingAuthenticationState>` component to cascading authentication state services

In .NET 7 or earlier, the `CascadingAuthenticationState` component is wrapped around some part of the UI tree, for example around the Blazor router, to provide cascading authentication state:

razor

```
<CascadingAuthenticationState>
    <Router ...>
    ...
</Router>
</CascadingAuthenticationState>
```

In .NET 8, don't use the `CascadingAuthenticationState` component:

diff

```
- <CascadingAuthenticationState>
    <Router ...>
    ...
```

```
</Router>  
- </CascadingAuthenticationState>
```

Instead, add cascading authentication state services to the service collection by calling [AddCascadingAuthenticationState](#) in the `Program` file:

```
C#  
  
builder.Services.AddCascadingAuthenticationState();
```

For more information, see the following resources:

- *ASP.NET Core Blazor authentication and authorization* article
 - [AuthenticationStateProvider](#) service
 - [Expose the authentication state as a cascading parameter](#)
 - [Customize unauthorized content with the Router component](#)
- [Secure ASP.NET Core server-side Blazor apps](#)

New article on HTTP caching issues

We've added a new article that discusses some of the common HTTP caching issues that can occur when upgrading Blazor apps across major versions and how to address HTTP caching issues.

For more information, see [Avoid HTTP caching issues when upgrading ASP.NET Core Blazor apps](#).

New article on class libraries with static server-side rendering (static SSR)

We've added a new article that discusses component library authorship in Razor class libraries (RCLs) with static server-side rendering (static SSR).

For more information, see [ASP.NET Core Razor class libraries \(RCLs\) with static server-side rendering \(static SSR\)](#).

Discover components from additional assemblies

When migrating from a Blazor Server app to a Blazor Web App, access the guidance in [ASP.NET Core Blazor routing and navigation](#) if the app uses routable components from additional assemblies, such as component class libraries.

Drop `[Parameter]` attribute when the parameter is supplied from a query string

The `[Parameter]` attribute is no longer required when supplying a parameter from the query string:

```
diff
```

```
- [Parameter]  
  [SupplyParameterFromQuery]
```

Blazor Server script fallback policy authorization

In .NET 7, the Blazor Server script (`blazor.server.js`) is [served by Static Files Middleware](#). Placing the call for Static Files Middleware (`UseStaticFiles`) in the request processing pipeline before the call to Authorization Middleware (`UseAuthorization`) is sufficient in .NET 7 apps to serve the Blazor script to anonymous users.

In .NET 8, the Blazor Server script is served [by its own endpoint](#), using endpoint routing. This change is introduced by [Fixed bug - Passing options to UseStaticFiles breaks Blazor Server \(dotnet/aspnetcore #45897\)](#).

Consider a multi-tenant scenario where:

- Both the default and fallback policies are set identically.
- The tenant is resolved using the first segment in the request path (for example, `tld.com/tenant-name/...`).
- The requests to tenant endpoints are authenticated by an additional authentication scheme, which adds an additional identity to the request principal.
- The fallback authorization policy has requirements that check claims via the additional identity.

Requests for the Blazor script file (`blazor.server.js`) are served at `/_framework/blazor.server.js`, which is hardcoded in the framework. Requests for the file aren't authenticated by the additional authentication scheme for tenants **but are still challenged by the fallback policy**, which results in returning an unauthorized result.

This problem is under evaluation for a new framework feature in [MapRazorComponents broken with FallbackPolicy RequireAuthenticatedUser \(dotnet/aspnetcore 51836\)](#), which is currently scheduled for .NET 9's release in November, 2024. Until then, you can work around this problem using any of the following three approaches:

- Don't use a fallback policy. Apply the `[Authorize]` attribute in the `_Imports.razor` file to apply it to all of the components of the app. For non-blazor endpoints, explicitly use `[Authorize]` or `RequireAuthorization`.
- Add `[AllowAnonymous]` to the `/_framework/blazor.server.js` endpoint in the `Program` file:

C#

```
app.MapBlazorHub().Add(endpointBuilder =>
{
    if (endpointBuilder is
        RouteEndpointBuilder
        {
            RoutePattern: { RawText: "/*_framework/blazor.server.js" }
        })
    {
        endpointBuilder.Metadata.Add(new AllowAnonymousAttribute());
    }
});
```

- Register a custom `AuthorizationHandler` that [checks the HttpContext](#) to allow the `/_framework/blazor.server.js` file through.

Docker

Update Docker images

For apps using Docker, update the *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 8.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 7.0 and 8.0:

diff

```
- docker pull mcr.microsoft.com/dotnet/aspnet:7.0
+ docker pull mcr.microsoft.com/dotnet/aspnet:8.0
```

Update Docker port

The default ASP.NET Core port configured in .NET container images has been updated from port 80 to 8080.

The new `ASPNETCORE_HTTP_PORTS` environment variable was added as a simpler alternative to `ASPNETCORE_URLS`.

For more information, see:

- [Default ASP.NET Core port changed from 80 to 8080.](#)
- [Specify ports only with ASPNETCORE_HTTP_PORTS](#)

Review breaking changes

For breaking changes from .NET Core .NET 7.0 to 8.0, see [Breaking changes in .NET 8](#), which includes [ASP.NET Core](#) and [Entity Framework Core](#) sections.

Migrate from ASP.NET Core 6.0 to 7.0

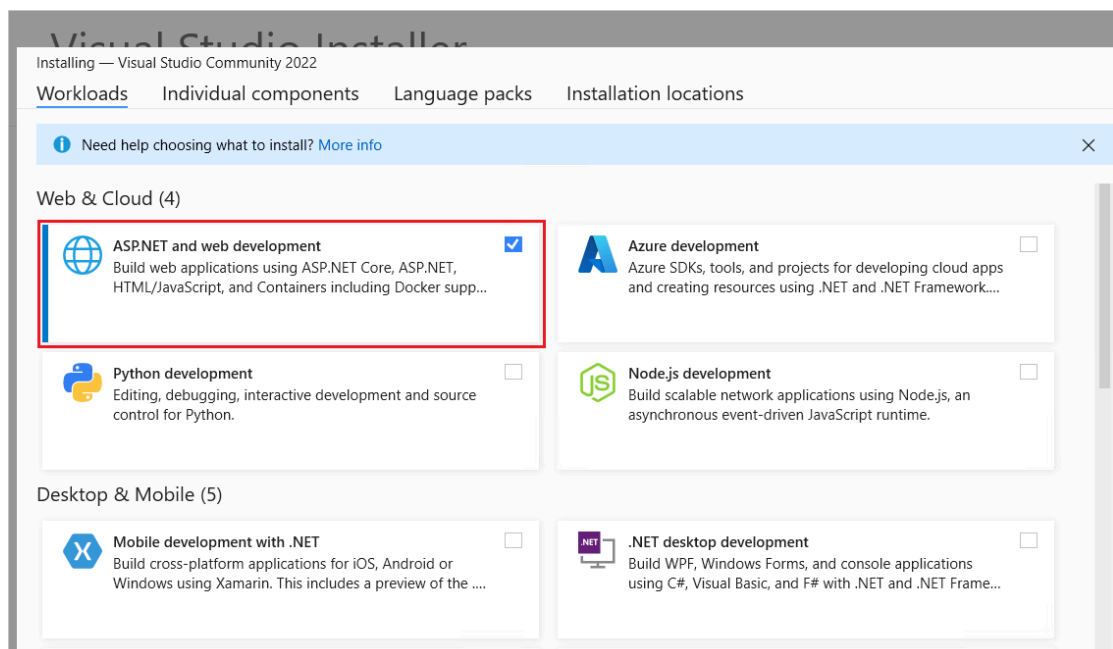
Article • 11/16/2024

This article explains how to update an existing ASP.NET Core 6.0 project to ASP.NET Core 7.0.

Prerequisites

Visual Studio

- [Visual Studio 2022](#) with the ASP.NET and web development workload.



Update .NET Core SDK version in global.json

If you rely on a [global.json](#) file to target a specific .NET Core SDK version, update the `version` property to the .NET 7.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
    - "version": "6.0.200"
    + "version": "7.0.100"
```

```
}  
}
```

Update the target framework

Update the project file's [Target Framework Moniker \(TFM\)](#) to `net7.0`:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Web">  
  
  <PropertyGroup>  
-    <TargetFramework>net6.0</TargetFramework>  
+    <TargetFramework>net7.0</TargetFramework>  
  </PropertyGroup>  
  
</Project>
```

Update package references

In the project file, update each [Microsoft.AspNetCore.*](#), [Microsoft.EntityFrameworkCore.*](#), [Microsoft.Extensions.*](#), and [System.Net.Http.Json](#) package reference's `Version` attribute to 7.0.0 or later. For example:

diff

```
<ItemGroup>  
-  <PackageReference Include="Microsoft.AspNetCore.JsonPatch"  
Version="6.0.9" />  
-  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"  
Version="6.0.9" />  
-  <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"  
Version="6.0.9" />  
-  <PackageReference Include="System.Net.Http.Json" Version="6.0.0" />  
+  <PackageReference Include="Microsoft.AspNetCore.JsonPatch"  
Version="7.0.0" />  
+  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"  
Version="7.0.0" />  
+  <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"  
Version="7.0.0" />  
+  <PackageReference Include="System.Net.Http.Json" Version="7.0.0" />  
</ItemGroup>
```


Blazor

Adopt .NET 7 features

After following the guidance earlier in this article to update an app to 7.0, adopt specific features by following the links in [What's new in ASP.NET Core 7.0](#).

To adopt all of the [new 7.0 features for Blazor apps](#), we recommend the following process:

- Create a new 7.0 Blazor project from one of the Blazor project templates. For more information, see [Tooling for ASP.NET Core Blazor](#).
- Move the app's components and code to the 7.0 app making modifications to adopt the new 7.0 features.

Simplify component parameter binding

In prior Blazor releases, binding across multiple components required binding to properties with `get`/`set` accessors.

In .NET 6 and earlier:

razor

```
<NestedGrandchild @bind-GrandchildMessage="BoundValue" />

@code {
    ...

    private string BoundValue
    {
        get => ChildMessage ?? string.Empty;
        set => ChildMessageChanged.InvokeAsync(value);
    }
}
```

In .NET 7, you can use the new `@bind:get` and `@bind:set` modifiers to support two-way data binding and simplify the binding syntax:

razor

```
<NestedGrandchild @bind-GrandchildMessage:get="ChildMessage"
    @bind-GrandchildMessage:set="ChildMessageChanged" />
```

For more information, see the following content in the *Data binding* article:

- [Introduction](#)
- [Bind across more than two components](#)

Migrate unmarshalled JavaScript interop

Unmarshalled interop using the [IJSUnmarshalledRuntime](#) interface is obsolete and should be replaced with JavaScript `[JSImport]` / `[JSExport]` interop.

For more information, see [JavaScript JSImport/JSExport interop with ASP.NET Core Blazor](#).

Blazor WebAssembly authentication uses history state for redirects

The support for authentication in Blazor WebAssembly apps changed to rely on [navigation history state](#) instead of query strings in the URL. As a result, passing the return URL through the query string fails to redirect back to the original page after a successful login in .NET 7.

The following example demonstrates **the prior redirection approach for apps that target .NET 6 or earlier** in `RedirectToLogin.razor`, which is based on a redirect URL (`?returnUrl=`) with [NavigateTo](#):

razor

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@code {
    protected override void OnInitialized()
    {
        Navigation.NavigateTo(
            $"authentication/login?returnUrl={
                Uri.EscapeDataString(Navigation.Uri)}");
    }
}
```

The following example demonstrates **the new redirection approach for apps that target .NET 7 or later** in `RedirectToLogin.razor`, which is based on [navigation history state](#) with [NavigateToLogin](#):

razor

```

@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Options

@inject
IOptionsSnapshot<RemoteAuthenticationOptions<ApiAuthorizationProviderOptions
>> OptionsSnapshot
@code {
    protected override void OnInitialized()
    {

        Navigation.NavigateToLogin(OptionsSnapshot.Get(Options.DefaultName).Authenti
        cationPaths.LogInPath);
    }
}

```

As part of this change, [SignOutSessionStateManager](#) is obsolete in .NET 7 or later and replaced with [NavigateToLogout](#).

The following example demonstrates **the prior approach** in `Shared/LoginDisplay.razor` of an app generated from the Blazor WebAssembly project template:

```

razor

@inject SignOutSessionStateManager SignOutManager

...

@code{
    private async Task BeginLogout(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }
}

```

The following example demonstrates **the new approach** in `Shared/LoginDisplay.razor` that calls [NavigateToLogout](#). The injection (`@inject`) of the [SignOutSessionStateManager](#) is removed from the component's directives at the top of the file, and the `BeginLogout` method is updated to the following code:

```

razor

@code{
    public void BeginLogout()
    {
        Navigation.NavigateToLogout("authentication/logout");
    }
}

```

```
}  
}
```

The `SignInSessionStateManager` service registration is removed in `Program.cs`:

diff

```
- builder.Services.AddScoped<SignInSessionStateManager>();
```

For more information, see the following resources:

- [\[Breaking change\]: Updates to Authentication in webassembly applications](#) ↗
- [ASP.NET Core Blazor routing and navigation](#)
- [Secure ASP.NET Core Blazor WebAssembly](#)
- [ASP.NET Core Blazor authentication and authorization](#)

.NET WebAssembly build tools for .NET 6 projects

You can now use the .NET WebAssembly build tools with a .NET 6 project when working with the .NET 7 SDK. The new `wasm-tools-net6` workload includes the .NET WebAssembly build tools for .NET 6 projects so that they can be used with the .NET 7 SDK. The existing `wasm-tools` workload installs the .NET WebAssembly build tools for .NET 7 projects. However, the .NET 7 version of the .NET WebAssembly build tools are incompatible with existing projects built with .NET 6. Projects using the .NET WebAssembly build tools that need to support both .NET 6 and .NET 7 must use multi-targeting.

Update Docker images

For apps using Docker, update your *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 7.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 6.0 and 7.0:

diff

```
- docker pull mcr.microsoft.com/dotnet/aspnet:6.0  
+ docker pull mcr.microsoft.com/dotnet/aspnet:7.0
```

Review breaking changes

For breaking changes from .NET Core 6.0 to .NET 7.0, see [Breaking changes in .NET 7](#).
ASP.NET Core and Entity Framework Core are included in the list.

Migrate from ASP.NET Core 5.0 to 6.0

Article • 10/18/2024

This article explains how to update an existing ASP.NET Core 5.0 project to ASP.NET Core 6.0. For instructions on how to migrate from ASP.NET Core 3.1 to ASP.NET Core 6.0, see [Migrate from ASP.NET Core 3.1 to 6.0](#).

Prerequisites

Visual Studio

- [Visual Studio 2022](#) with the **ASP.NET and web development** workload.
- [.NET 6.0 SDK](#)

Update .NET SDK version in global.json

If you rely upon a [global.json](#) file to target a specific .NET SDK version, update the `version` property to the .NET 6.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "5.0.100"
+    "version": "6.0.100"
  }
}
```

Update the target framework

Update the project file's [Target Framework Moniker \(TFM\)](#) to `net6.0`:

diff

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

  <PropertyGroup>
-    <TargetFramework>net5.0</TargetFramework>
+    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
```

```
</Project>
```

Update package references

In the project file, update each [Microsoft.AspNetCore.*](#) and [Microsoft.Extensions.*](#) package reference's `Version` attribute to 6.0.0 or later. For example:

diff

```
<ItemGroup>
-   <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="5.0.3" />
-   <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="5.0.0" />
+   <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="6.0.0" />
+   <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="6.0.0" />
</ItemGroup>
```

New hosting model

The new .NET 6 minimal hosting model for ASP.NET Core apps requires only one file and a few lines of code. **Apps migrating to 6.0 don't need to use the new minimal hosting model.** For more information, see [Apps migrating to 6.0 don't need to use the new minimal hosting model](#) in the following section.

The following code from the ASP.NET Core empty template creates an app using the new minimal hosting model:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

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

app.Run();
```

The minimal hosting model:

- Significantly reduces the number of files and lines of code required to create an app. Only one file is needed with four lines of code.

- Unifies `Startup.cs` and `Program.cs` into a single `Program.cs` file.
- Uses [top-level statements](#) to minimize the code required for an app.
- Uses [global using directives](#) to eliminate or minimize the number of `using` [statement](#) lines required.

The following code displays the `Startup.cs` and `Program.cs` files from an ASP.NET Core 5 Web App template (Razor Pages) with unused `using` statements removed:

C#

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
// Unused usings removed.

namespace WebAppRPv5
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add
        // services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddRazorPages();
        }

        // This method gets called by the runtime. Use this method to
        // configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                // The default HSTS value is 30 days. You may want to change
                // this for production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }
        }
    }
}
```



```

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

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}

```

C#

```

using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
// Unused usings removed.

namespace WebAppRPv5
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

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

```

In ASP.NET Core 6, the preceding code is replaced by the following:

C#

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())

```

```

{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
    // production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

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

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();

```

The preceding ASP.NET Core 6 sample shows how:

- `ConfigureServices` is replaced with `WebApplication.Services`.
- `builder.Build()` returns a configured `WebApplication` to the variable `app`.
`Configure` is replaced with configuration calls to same services using `app`.

Detailed examples of migrating ASP.NET Core 5 `Startup` code to ASP.NET Core 6 using the minimal hosting model are provided later in this document.

There are a few changes to the other files generated for the Web App template:

- `Index.cshtml` and `Privacy.cshtml` have the unused `using` statements removed.
- `RequestId` in `Error.cshtml` is declared as a [nullable reference type \(NRT\)](#):

diff

```

- public string RequestId { get; set; }
+ public string? RequestId { get; set; }

```

- Log level defaults have changed in `appsettings.json` and `appsettings.Development.json`:

diff

```

- "Microsoft": "Warning",
- "Microsoft.Hosting.Lifetime": "Information"
+ "Microsoft.AspNetCore": "Warning"

```

In the preceding ASP.NET Core template code, `"Microsoft": "Warning"` has been changed to `"Microsoft.AspNetCore": "Warning"`. This change results in logging all informational messages from the `Microsoft` namespace *except* `Microsoft.AspNetCore`. For example, `Microsoft.EntityFrameworkCore` is now logged at the informational level.

For more details on the new hosting model, see the [Frequently asked questions](#) section. For more information on the adoption of NRTs and .NET compiler null-state analysis, see the [Nullable reference types \(NRTs\) and .NET compiler null-state static analysis](#) section.

Apps migrating to or using 6.0 and later don't need to use the new minimal hosting model

Using [Startup](#) and the [Generic Host](#) used by the ASP.NET Core 3.1 and 5.0 templates is fully supported.

Use Startup with the new minimal hosting model

ASP.NET Core 3.1 and 5.0 apps can use their `Startup` code with the new minimal hosting model. Using `Startup` with the minimal hosting model has the following advantages:

- No hidden reflection is used to call the `Startup` class.
- Asynchronous code can be written because the developer controls the call to `Startup`.
- Code can be written that interleaves `ConfigureServices` and `Configure`.

One minor limitation in using `Startup` code with the new minimal hosting model is that to inject a dependency into `Configure`, the service in `Program.cs` must be manually resolved.

Consider the following code generated by the ASP.NET Core 3.1 or 5.0 Razor Pages template:

C#

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

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

```

        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
    }

```

C#

```

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

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

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

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}

```

The preceding code migrated to the new minimal hosting model:

C#

```

using Microsoft.AspNetCore.Builder;

var builder = WebApplication.CreateBuilder(args);

```

```
var startup = new Startup(builder.Configuration);

startup.ConfigureServices(builder.Services);

var app = builder.Build();

startup.Configure(app, app.Environment);

app.Run();
```

C#

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

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

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

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

In the preceding code, the `if (env.IsDevelopment())` block is removed because in [development mode](#), the developer exception page middleware is enabled by default. For more information, see [Differences between the ASP.NET Core 5 and 6 hosting models](#) in the next section.

When using a custom dependency injection (DI) container, add the following highlighted code:

C#

```
using Autofac;
using Autofac.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.Hosting;

var builder = WebApplication.CreateBuilder(args);

var startup = new Startup(builder.Configuration);

startup.ConfigureServices(builder.Services);

// Using a custom DI container.
builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());
builder.Host.ConfigureContainer<ContainerBuilder>
(startup.ConfigureContainer);

var app = builder.Build();

startup.Configure(app, app.Environment);

app.Run();
```

C#

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

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    // Using a custom DI container
    public void ConfigureContainer(ContainerBuilder builder)
    {
        // Configure custom container.
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (!env.IsDevelopment())
```

```

    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
}

```

When using the minimal hosting model, the endpoint routing middleware wraps the entire middleware pipeline, therefore there's no need to have explicit calls to `UseRouting` or `UseEndpoints` 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.

In the following code, the calls to `UseRouting` and `UseEndpoints` are removed from `Startup`. `MapRazorPages` is called in `Program.cs`:

C#

```

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

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

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

        app.UseHttpsRedirection();
        app.UseStaticFiles();
    }
}

```

```

        //app.UseRouting();

        //app.UseEndpoints(endpoints =>
        //{
        //    endpoints.MapRazorPages();
        //});
    }
}

```

C#

```

using Microsoft.AspNetCore.Builder;

var builder = WebApplication.CreateBuilder(args);

var startup = new Startup(builder.Configuration);

startup.ConfigureServices(builder.Services);

var app = builder.Build();

startup.Configure(app, app.Environment);

app.MapRazorPages();

app.Run();

```

When using `Startup` with the new minimal hosting model, keep in mind the following difference:

- `Program.cs` controls the instantiation and lifetime of the `Startup` class.
- Any additional services injected into the `Configure` method need to be manually resolved by the `Program` class.

Differences between the ASP.NET Core 5 and 6 hosting models

- In [development mode](#), the developer exception page middleware is enabled by default.
- The app name defaults to the entry point assembly's name: `Assembly.GetEntryAssembly().GetName().FullName`. When using the [WebApplicationBuilder](#) in a library, explicitly change the app name to the library's assembly to allow MVC's [application part discovery](#) to work. See [Change the content root, app name, and environment](#) in this document for detailed instructions.

- The endpoint routing middleware wraps the entire middleware pipeline, therefore there's no need to have explicit calls to `UseRouting` or `UseEndpoints` 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.
- The [pipeline](#) is created before any `IStartupFilter` runs, therefore exceptions caused while building the pipeline aren't visible to the `IStartupFilter` call chain.
- Some tools, such as EF migrations, use `Program.CreateHostBuilder` to access the app's `IServiceProvider` to execute custom logic in the context of the app. These tools have been updated to use a new technique to execute custom logic in the context of the app. [Entity Framework Migrations](#) is an example of a tool that uses `Program.CreateHostBuilder` in this way. We're working to make sure tools are updated to use the new model.
- Unlike the `Startup` class, the minimal host doesn't automatically configure a DI scope when instantiating the service provider. For contexts where a scope is required, it is necessary to invoke `IServiceScope` with `IServiceScopeFactory.CreateScope` to instantiate a new scope. For more information, see [how to resolve a service at app startup](#).
- It's **not** possible to [change any host settings such as app name, environment, or the content root](#) after the creation of the `WebApplicationBuilder`. For detailed instructions on changing host settings, see [Customize IHostBuilder or IWebHostBuilder](#). The following highlighted APIs throw an exception:

C#

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

builder.Services.AddRazorPages();

// WebHost
```

```
try
{
    builder.WebHost.UseContentRoot(Directory.GetCurrentDirectory());
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

try
{
    builder.WebHost.UseEnvironment(Environments.Staging);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

try
{
    builder.WebHost.UseSetting(WebHostDefaults.ApplicationKey,
    "ApplicationName2");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

try
{
    builder.WebHost.UseSetting(WebHostDefaults.ContentRootKey,
    Directory.GetCurrentDirectory());
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

try
{
    builder.WebHost.UseSetting(WebHostDefaults.EnvironmentKey,
    Environments.Staging);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

// Host
try
{
    builder.Host.UseEnvironment(Environments.Staging);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

```

        Console.WriteLine(ex.Message);
    }

    try
    {
        // TODO: This does not throw
        builder.Host.UseContentRoot(Directory.GetCurrentDirectory());
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }

    var app = builder.Build();

    if (!app.Environment.IsDevelopment())
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

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

    app.UseRouting();

    app.UseAuthorization();

    app.MapRazorPages();

    app.Run();

```

- The `Startup` class can't be used from [WebApplicationBuilder.Host](#) or [WebApplicationBuilder.WebHost](#). The following highlighted code throws an exception:

```

C#

var builder = WebApplication.CreateBuilder(args);

try
{
    builder.Host.ConfigureWebHostDefaults(webHostBuilder =>
    {
        webHostBuilder.UseStartup<Startup>();
    });
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    throw;
}

```

```
builder.Services.AddRazorPages();

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

C#

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

try
{
    builder.WebHost.UseStartup<Startup>();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    throw;
}

builder.Services.AddRazorPages();

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

- The [IHostBuilder](#) implementation on [WebApplicationBuilder](#) (`WebApplicationBuilder.Host`), doesn't defer execution of the [ConfigureServices](#), [ConfigureAppConfiguration](#), or [ConfigureHostConfiguration](#) methods. Not deferring execution allows code using [WebApplicationBuilder](#) to observe changes made to the `IServiceCollection` and `IConfiguration`. The following example only adds `Service1` as an `IService`:

C#

```
using Microsoft.Extensions.DependencyInjection.Extensions;

var builder = WebApplication.CreateBuilder(args);

builder.Host.ConfigureServices(services =>
{
    services.TryAddSingleton<IService, Service1>();
});

builder.Services.TryAddSingleton<IService, Service2>();

var app = builder.Build();

// Displays Service1 only.
Console.WriteLine(app.Services.GetRequiredService<IService>());

app.Run();

class Service1 : IService
```

```
{  
}  
  
class Service2 : IService  
{  
}  
  
interface IService  
{  
}
```

In the preceding code, the `builder.Host.ConfigureServices` callback gets called inline rather than being deferred until `builder.Build` is called. This means that `Service1` gets added to the `IServiceCollection` before `Service2` and results in `Service1` being resolved for `IService`.

Building libraries for ASP.NET Core 6

The existing .NET ecosystem built extensibility around [IServiceCollection](#), [IHostBuilder](#), and [IWebHostBuilder](#). These properties are available on [WebApplicationBuilder](#) as `Services`, `Host`, and `WebHost`.

`WebApplication` implements both [Microsoft.AspNetCore.Builder.IApplicationBuilder](#) and [Microsoft.AspNetCore.Routing.IEndpointRouteBuilder](#).

We expect library authors to continue targeting `IHostBuilder`, `IWebHostBuilder`, `IApplicationBuilder`, and `IEndpointRouteBuilder` when building ASP.NET Core specific components. This ensures that your middleware, route handler, or other extensibility points continue to work across different hosting models.

Frequently asked questions (FAQ)

- Is the new minimal hosting model less capable?

No. The new hosting model is functionally equivalent for 98% of scenarios supported by `IHostBuilder` and the `IWebHostBuilder`. There are some advanced scenarios that require specific workarounds on `IHostBuilder`, but we expect those to be extremely rare.

- Is the generic hosting model deprecated?

No. The generic hosting model is an alternative model that is supported indefinitely. The generic host underpins the new hosting model and is still the

primary way to host worker-based applications.

- **Do I have to migrate to the new hosting model?**

No. The new hosting model is the preferred way to host new apps using .NET 6 and later, but you aren't forced to change the project layout in existing apps. This means apps can upgrade from .NET 5 to .NET 6 by changing the target framework in the project file from `net5.0` to `net6.0`. For more information, see the [Update the target framework](#) section in this article. However, we recommend apps migrate to the new hosting model to take advantage of new features only available to the new hosting model.

- **Do I have to use top-level statements?**

No. The new project templates all use [top-level statements](#), but the new hosting APIs can be used in any .NET 6 app to host a webserver or web app.

- **Where do I put state that was stored as fields in my `Program` or `Startup` class?**

We strongly recommend using [dependency injection](#) (DI) to flow state in ASP.NET Core apps.

There are two approaches to storing state outside of DI:

- Store the state in another class. Storing in a class assumes a static state that can be accessed from anywhere in the app.
- Use the `Program` class generated by top level statements to store state. Using `Program` to store state is the semantic approach:

C#

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

ConfigurationValue = builder.Configuration["SomeKey"] ?? "Hello";

var app = builder.Build();

app.MapGet("/", () => ConfigurationValue);

app.Run();

partial class Program
{
    public static string? ConfigurationValue { get; private set; }
}
```

- What if I was using a custom dependency injection container?

Custom DI containers are supported. For an example, see [Custom dependency injection \(DI\) container](#).

- Do `WebApplicationFactory` and `TestServer` still work?

Yes. `WebApplicationFactory<TEntryPoint>` is the way to test the new hosting model.

For an example, see [Test with WebApplicationFactory or TestServer](#).

Blazor

After following the guidance earlier in this article to update an app to 6.0, adopt specific features by following the links in [What's new in ASP.NET Core 6.0](#).

To adopt all of the [new 6.0 features for Blazor apps](#), we recommend the following process:

- Create a new 6.0 Blazor project from one of the Blazor project templates. For more information, see [Tooling for ASP.NET Core Blazor](#).
- Move the app's components and code to the 6.0 app making modifications to adopt the new 6.0 features.

Migrating SPA projects

Migrating Angular apps from SPA extensions

See [this GitHub issue](#) ↗

Migrating React apps from SPA extensions

See Migrating React applications from Spa Extensions in [this GitHub issue](#) ↗

Update Docker images

For apps using Docker, update your *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 6.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 5.0 and 6.0:

```
diff
```

```
- docker pull mcr.microsoft.com/dotnet/aspnet:5.0
+ docker pull mcr.microsoft.com/dotnet/aspnet:6.0
```

See GitHub issue [Breaking Change: Default console logger format set to JSON](#).

Changes to the ASP.NET Core Razor SDK

The Razor compiler now leverages the new [source generators feature](#) to generate compiled C# files from the Razor views and pages in a project. In previous versions:

- The compilation relied on the `RazorGenerate` and `RazorCompile` targets to produce the generated code. These targets are no longer valid. In .NET 6, both code generation and compilation are supported by a single call to the compiler. `RazorComponentGenerateDependsOn` is still supported to specify dependencies that are required before the build runs.
- A separate Razor assembly, `AppName.Views.dll`, was generated that contained the compiled view types in an application. This behavior has been deprecated and a single assembly `AppName.dll` is produced that contains both the app types and the generated views.
- The app types in `AppName.Views.dll` were public. In .NET 6, the app types are in `AppName.dll` but are `internal sealed`. Apps doing type discover on `AppName.Views.dll` won't be able to do type discover on `AppName.dll`. The following shows the API change:

diff

```
- public class Views_Home_Index :
global::Microsoft.AspNetCore.Mvc.Razor.RazorPage<dynamic>
+ internal sealed class Views_Home_Index :
global::Microsoft.AspNetCore.Mvc.Razor.RazorPage<dynamic>
```

Make the following changes:

- The following properties are no longer applicable with the single-step compilation model.
 - `RazorTargetAssemblyAttribute`
 - `RazorTargetName`
 - `EnableDefaultRazorTargetAssemblyInfoAttributes`
 - `UseRazorBuildServer`
 - `GenerateRazorTargetAssemblyInfo`
 - `GenerateMvcApplicationPartsAssemblyAttributes`

For more information, see [Razor compiler no longer produces a Views assembly](#).

Project templates use Duende Identity Server

Project templates now use [Duende Identity Server](#).

📌 Important

Duende Identity Server is an open source product with a reciprocal license agreement. If you plan to use Duende Identity Server in production, you might be required to obtain a commercial licence from [Duende Software](#) and pay a license fee. For more information, see [Duende Software: Licenses](#).

To learn how to use [Microsoft Azure Active Directory](#) for ASP.NET Core Identity, see [Identity_\(dotnet/aspnetcore GitHub repository\)](#).

Add a `DbSet<Key>` property named `Keys` to every `IdentityDbContext` to satisfy a new requirement from the updated version of `IPersistedGrantDbContext`. The keys are required as part of the contract with Duende Identity Server's stores.

C#

```
public DbSet<Key> Keys { get; set; }
```

📌 Note

Existing migrations must be recreated for Duende Identity Server.

Code samples migrated to ASP.NET Core 6.0

[Code samples migrated to the new minimal hosting model in 6.0](#)

Review breaking changes

See the following resources:

- [Identity: Default Bootstrap version of UI changed](#)
- [Breaking changes for migration from version 5.0 to 6.0](#): Includes ASP.NET Core and Entity Framework Core.

- [Announcements GitHub repository \(aspnet/Announcements, 6.0.0 label\) ↗](#):
Includes breaking and non-breaking information.

Nullable reference types (NRTs) and .NET compiler null-state static analysis

ASP.NET Core project templates use nullable reference types (NRTs), and the .NET compiler performs null-state static analysis. These features were released with C# 8 and are enabled by default for apps generated using ASP.NET Core 6.0 (C# 10) or later.

The .NET compiler's null-state static analysis warnings can either serve as a guide for updating a documentation example or sample app locally or be ignored. Null-state static analysis can be disabled by [setting Nullable to disable](#) in the app's project file, which we only recommend for documentation examples and sample apps if the compiler warnings are distracting while learning about .NET. ***We don't recommend disabling null-state checking in production projects.***

For more information on NRTs, the MSBuild `Nullable` property, and updating apps (including `#pragma` guidance), see the following resources in the C# documentation:

- [Nullable reference types](#)
- [Nullable reference types \(C# reference\)](#)
- [Learn techniques to resolve nullable warnings](#)
- [Update a codebase with nullable reference types to improve null diagnostic warnings](#)
- [Attributes for null-state static analysis](#)
- [! \(null-forgiving\) operator \(C# reference\)](#)

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](#).

Application name change

In .NET 6, [WebApplicationBuilder](#) normalizes the content root path to end with a [DirectorySeparatorChar](#). Most apps migrating from [HostBuilder](#) or [WebHostBuilder](#)

won't have the same app name because they aren't normalized. For more information, see [SetApplicationName](#)

Additional resources

- [Code samples migrated to the new minimal hosting model in 6.0](#)

Code samples migrated to the new minimal hosting model in ASP.NET Core 6.0

Article • 07/14/2023

This article provides samples of code migrated to ASP.NET Core 6.0. ASP.NET Core 6.0 uses a new minimal hosting model. For more information, see [New hosting model](#).

Middleware

The following code adds the Static File Middleware to an ASP.NET Core 5 app:

C#

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseStaticFiles();
    }
}
```

The following code adds the Static File Middleware to an ASP.NET Core 6 app:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseStaticFiles();

app.Run();
```

[WebApplication.CreateBuilder](#) initializes a new instance of the [WebApplicationBuilder](#) class with preconfigured defaults. For more information, see [ASP.NET Core Middleware](#)

Routing

The following code adds an endpoint to an ASP.NET Core 5 app:

C#

```

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", () => "Hello World");
        });
    }
}

```

In .NET 6, routes can be added directly to the [WebApplication](#) without an explicit call to [UseEndpoints](#) or [UseRouting](#). The following code adds an endpoint to an ASP.NET Core 6 app:

C#

```

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

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

app.Run();

```

Note: Routes added directly to the [WebApplication](#) execute at the *end* of the pipeline.

Change the content root, app name, and environment

ASP.NET Core 5

C#

```

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseEnvironment(Environments.Staging)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>()
                .UseSetting(WebHostDefaults.ApplicationKey,
                    typeof(Program).Assembly.FullName);
        });

```

ASP.NET Core 6

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    ApplicationName = typeof(Program).Assembly.FullName,
    ContentRootPath = Directory.GetCurrentDirectory(),
    EnvironmentName = Environments.Staging,
    WebRootPath = "customwwwroot"
});

Console.WriteLine($"Application Name:
{builder.Environment.ApplicationName}");
Console.WriteLine($"Environment Name:
{builder.Environment.EnvironmentName}");
Console.WriteLine($"ContentRoot Path:
{builder.Environment.ContentRootPath}");
Console.WriteLine($"WebRootPath: {builder.Environment.WebRootPath}");

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

For more information, see [ASP.NET Core fundamentals overview](#)

Change the content root, app name, and environment by environment variables or command line

The following table shows the environment variable and command-line argument used to change the content root, app name, and environment:

 Expand table

feature	Environment variable	Command-line argument
Application name	ASPNETCORE_APPLICATIONNAME	--applicationName
Environment name	ASPNETCORE_ENVIRONMENT	--environment
Content root	ASPNETCORE_CONTENTROOT	--contentRoot

Add configuration providers

ASP.NET Core 5

C#

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

ASP.NET Core 6

C#

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

builder.Configuration.AddIniFile("appsettings.ini");

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

For detailed information, see [File configuration providers](#) in [Configuration in ASP.NET Core](#).

Add logging providers

ASP.NET Core 5

C#

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

ASP.NET Core 6

C#

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

// Configure JSON logging to the console.
builder.Logging.AddJsonConsole();

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

For more information, see [Logging in .NET Core and ASP.NET Core](#).

Add services

ASP.NET Core 5

```
C#

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add the memory cache services
        services.AddMemoryCache();

        // Add a custom scoped service
        services.AddScoped<ITodoRepository, TodoRepository>();
    }
}
```

ASP.NET Core 6

```
C#

var builder = WebApplication.CreateBuilder(args);

// Add the memory cache services.
builder.Services.AddMemoryCache();

// Add a custom scoped service.
builder.Services.AddScoped<ITodoRepository, TodoRepository>();
var app = builder.Build();
```

For more information, see [Dependency injection in ASP.NET Core](#).

Customize IHostBuilder or IWebHostBuilder

Customize IHostBuilder

ASP.NET Core 5

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureHostOptions(o => o.ShutdownTimeout =
            TimeSpan.FromSeconds(30));
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

ASP.NET Core 6

C#

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

// Wait 30 seconds for graceful shutdown.
builder.Host.ConfigureHostOptions(o => o.ShutdownTimeout =
    TimeSpan.FromSeconds(30));

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

Customize IWebHostBuilder

ASP.NET Core 5

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            // Change the HTTP server implementation to be HTTP.sys based.
            webBuilder.UseHttpSys()
                .UseStartup<Startup>();
        });
```

ASP.NET Core 6

C#

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

// Change the HTTP server implementation to be HTTP.sys based.
// Windows only.
builder.WebHost.UseHttpSys();

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

Change the web root

By default, the web root is relative to the content root in the `wwwroot` folder. Web root is where the static files middleware looks for static files. Web root can be changed by setting the [WebRootPath](#) property on [WebApplicationOptions](#):

ASP.NET Core 5

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            // Look for static files in webroot.
            webBuilder.UseWebRoot("webroot")
                .UseStartup<Startup>();
        });
```

ASP.NET Core 6

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    // Look for static files in webroot
    WebRootPath = "webroot"
});

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

Custom dependency injection (DI) container

The following .NET 5 and .NET 6 samples use [Autofac](#)

ASP.NET Core 5

Program class

C#

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

Startup

C#

```
public class Startup
{
    public void ConfigureContainer(ContainerBuilder containerBuilder)
    {
    }
}
```

ASP.NET Core 6

C#

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

builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());

// Register services directly with Autofac here. Don't
// call builder.Populate(), that happens in AutofacServiceProviderFactory.
builder.Host.ConfigureContainer<ContainerBuilder>(builder =>
    builder.RegisterModule(new MyApplicationModule()));

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

Access additional services

`Startup.Configure` can inject any service added via the [IServiceCollection](#).

ASP.NET Core 5

C#

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add
    // services
    // to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IService, Service>();
    }

    // Anything added to the service collection can be injected into
    // Configure.
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env,
        IHostApplicationLifetime lifetime,
        IService service,
        ILogger<Startup> logger)
    {
        lifetime.ApplicationStarted.Register(() =>
            logger.LogInformation(
                "The application {Name} started in the injected {Service}",
                env.ApplicationName, service));
    }
}
```

ASP.NET Core 6

In ASP.NET Core 6:

- There are a few common services available as top level properties on [WebApplication](#).
- Additional services need to be manually resolved from the `IServiceProvider` via [WebApplication.Services](#).

C#

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

builder.Services.AddSingleton<IService, Service>();

var app = builder.Build();

IService service = app.Services.GetRequiredService<IService>();
ILogger logger = app.Logger;
IHostApplicationLifetime lifetime = app.Lifetime;
```

```
IWebHostEnvironment env = app.Environment;

lifetime.ApplicationStarted.Register(() =>
    logger.LogInformation(
        $"The application {env.ApplicationName} started" +
        $" with injected {service}"));
```

Test with WebApplicationFactory or TestServer

ASP.NET Core 5

In the following samples, the test project uses `TestServer` and `WebApplicationFactory<TEntryPoint>`. These ship as separate packages that require explicit reference:

WebApplicationFactory

XML

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.Mvc.Testing" Version="{Version}" />
</ItemGroup>
```

TestServer

XML

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.TestHost" Version="{Version}" />
</ItemGroup>
```

ASP.NET Core 5 code

C#

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IHelloService, HelloService>();
    }
}
```

```

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
        IHelloService helloService)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await
context.Response.WriteAsync(helloService.HelloMessage);
            });
        });
    }
}

```

With TestServer

C#

```

[Fact]
public async Task HelloWorld()
{
    using var host = Host.CreateDefaultBuilder()
        .ConfigureWebHostDefaults(builder =>
        {
            // Use the test server and point to the application's startup
            builder.UseTestServer()
                .UseStartup<WebApplication1.Startup>();
        })
        .ConfigureServices(services =>
        {
            // Replace the service
            services.AddSingleton<IHelloService, MockHelloService>();
        })
        .Build();

    await host.StartAsync();

    var client = host.GetTestClient();

    var response = await client.GetStringAsync("/");

    Assert.Equal("Test Hello", response);
}

```

```
class MockHelloService : IHelloService
{
    public string HelloMessage => "Test Hello";
}
```

With WebApplicationFactory

C#

```
[Fact]
public async Task HelloWorld()
{
    var application = new WebApplicationFactory<Program>()
        .WithWebHostBuilder(builder =>
        {
            builder.ConfigureServices(services =>
            {
                services.AddSingleton<IHelloService, MockHelloService>();
            });
        });

    var client = application.CreateClient();

    var response = await client.GetStringAsync("/");

    Assert.Equal("Test Hello", response);
}

class MockHelloService : IHelloService
{
    public string HelloMessage => "Test Hello";
}
```

ASP.NET Core 6

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IHelloService, HelloService>();

var app = builder.Build();

var helloService = app.Services.GetRequiredService<IHelloService>();

app.MapGet("/", async context =>
{
    await context.Response.WriteAsync(helloService.HelloMessage);
});
```

```
app.Run();
```

Project file (.csproj)

The project file can contain one of the following:

XML

```
<ItemGroup>
  <InternalsVisibleTo Include="MyTestProject" />
</ItemGroup>
```

Or

```
[assembly: InternalsVisibleTo("MyTestProject")]
```

An alternative solution is to make the `Program` class public. `Program` can be made public with [Top-level statements](#) by defining a `public partial Program` class in the project or in `Program.cs`:

C#

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

// ... Configure services, routes, etc.

app.Run();

public partial class Program { }
```

C#

```
[Fact]
public async Task HelloWorld()
{
    var application = new WebApplicationFactory<Program>()
        .WithWebHostBuilder(builder =>
        {
            builder.ConfigureServices(services =>
            {
                services.AddSingleton<IHelloService, MockHelloService>();
            });
        });
}
```



```
var client = application.CreateClient();

var response = await client.GetStringAsync("/");

Assert.Equal("Test Hello", response);
}

class MockHelloService : IHelloService
{
    public string HelloMessage => "Test Hello";
}
```

The .NET 5 version and .NET 6 version with the `WebApplicationFactory` are identical by design.

Migrate from ASP.NET Core 3.1 to 6.0

Article • 06/04/2022

This article explains how to update an existing ASP.NET Core 3.1 project to ASP.NET Core 6.0. To upgrade from ASP.NET Core 5.0 to 6.0, see [Migrate from ASP.NET Core 5.0 to 6.0](#).

Prerequisites

Visual Studio

- [Visual Studio 2022](#) [↗] with the **ASP.NET and web development** workload.
- [.NET 6.0 SDK](#) [↗]

Update .NET SDK version in `global.json`

If you rely upon a `global.json` file to target a specific .NET SDK version, update the `version` property to the .NET 6.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "3.1.200"
+    "version": "6.0.100"
  }
}
```

Update the target framework

Update the project file's **Target Framework Moniker (TFM)** to `net6.0`:

diff

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

  <PropertyGroup>
-    <TargetFramework>netcoreapp3.1</TargetFramework>
+    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
```

```
</Project>
```

Update package references

In the project file, update each [Microsoft.AspNetCore.*](#), [Microsoft.EntityFrameworkCore.*](#), [Microsoft.Extensions.*](#), and [System.Net.Http.Json](#) package reference's `Version` attribute to 6.0.0 or later. For example:

```
diff
```

```
<ItemGroup>
-   <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="3.1.6" />
-   <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="3.1.6" />
-   <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="3.1.6" />
-   <PackageReference Include="System.Net.Http.Json" Version="3.2.1" />
+   <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="6.0.0" />
+   <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="6.0.0" />
+   <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="6.0.0" />
+   <PackageReference Include="System.Net.Http.Json" Version="6.0.0" />
</ItemGroup>
```

Delete `bin` and `obj` folders

You may need to delete the `bin` and `obj` folders. Run `dotnet nuget locals --clear all` to clear the NuGet package cache.

Minimal hosting model

The ASP.NET Core templates generate code using the new [minimal hosting model](#). The minimal hosting model unifies `Startup.cs` and `Program.cs` into a single `Program.cs` file. `ConfigureServices` and `Configure` are no longer used. Apps migrating from ASP.NET Core 3.1 to 6.0 don't need to use the minimal hosting model, using `Startup` and the [Generic Host](#) used by the ASP.NET Core 3.1 templates is fully supported.

To use `Startup` with the new minimal hosting model, see [Use Startup with the new minimal hosting model](#).

To migrate to the new minimal hosting model using the following pattern used by the ASP.NET Core 6.0 templates, see [Code samples migrated to the new minimal hosting model in ASP.NET Core 6.0](#) and [Migrate from ASP.NET Core 5.0 to 6.0](#)

Update Razor class libraries (RCLs)

Migrate Razor class libraries (RCLs) to take advantage of new APIs or features that are introduced as part of ASP.NET Core 6.0.

To update a RCL that targets components:

1. Update the following properties in the project file:

```
diff

<Project Sdk="Microsoft.NET.Sdk.Razor">
  <PropertyGroup>
-    <TargetFramework>netstandard2.0</TargetFramework>
-    <RazorLangVersion>3.0</RazorLangVersion>
+    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
```

2. Update other packages to their latest versions. The latest versions can be found at [NuGet.org](#).

To update an RCL targeting MVC, update the following properties in the project file:

```
diff

<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
-    <TargetFramework>netcoreapp3.1</TargetFramework>
+    <TargetFramework>net6.0</TargetFramework>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>
```

Blazor

To adopt all of the [5.0 features](#) and [6.0 features](#) for Blazor apps, we recommend the following process:

- Create a new 6.0 Blazor project from one of the Blazor project templates. For more information, see [Tooling for ASP.NET Core Blazor](#).

- Move the app's components and code to the 6.0 app making modifications to adopt the new 5.0 and 6.0 features.

Update Docker images

For apps using Docker, update your *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 6.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 3.1 and 6.0:

diff

```
- docker pull mcr.microsoft.com/dotnet/core/aspnet:3.1
+ docker pull mcr.microsoft.com/dotnet/aspnet:6.0
```

As part of the move to ".NET" as the product name, the Docker images moved from the `mcr.microsoft.com/dotnet/core` repositories to `mcr.microsoft.com/dotnet`. For more information, see [.NET 5.0 - Docker Repo Name Change \(dotnet/dotnet-docker #1939\)](#).

Model binding changes in ASP.NET Core MVC and Razor Pages

`DateTime` values are model bound as UTC times

In ASP.NET Core 3.1 and earlier, `DateTime` values were model-bound as local time, where the timezone was determined by the server. `DateTime` values bound from input formatting (JSON) and `DateTimeOffset` values were bound as UTC timezones.

In ASP.NET Core 5.0 and later, model binding consistently binds `DateTime` values with the UTC timezone.

To retain the previous behavior, remove the `DateTimeModelBinderProvider` in `Startup.ConfigureServices`:

C#

```
services.AddControllersWithViews(options =>
    options.ModelBinderProviders.RemoveType<DateTimeModelBinderProvider>());
```

ComplexObjectModelBinderProvider ** **ComplexObjectModelBinder **replace** **ComplexTypeModelBinderProvider ** **ComplexTypeModelBinder**

To add support for model binding [C# 9 record types](#), the [ComplexTypeModelBinderProvider](#) is:

- Annotated as obsolete.
- No longer registered by default.

Apps that rely on the presence of the `ComplexTypeModelBinderProvider` in the `ModelBinderProviders` collection need to reference the new binder provider:

diff

```
- var complexModelBinderProvider =  
options.ModelBinderProviders.OfType<ComplexTypeModelBinderProvider>();  
+ var complexModelBinderProvider =  
options.ModelBinderProviders.OfType<ComplexObjectModelBinderProvider>();
```

UseDatabaseErrorPage **obsolete**

The ASP.NET Core 3.1 templates that include an option for individual user accounts generate a call to [UseDatabaseErrorPage](#). `UseDatabaseErrorPage` is now obsolete and should be replaced with a combination of `AddDatabaseDeveloperPageExceptionFilter` and `UseMigrationsEndPoint`, as shown in the following code:

diff

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddDbContext<ApplicationDbContext>(options =>  
        options.UseSqlServer(  
            Configuration.GetConnectionString("DefaultConnection")));  
+    services.AddDatabaseDeveloperPageExceptionFilter();  
    services.AddDefaultIdentity<IdentityUser>(options =>  
options.SignIn.RequireConfirmedAccount = true)  
        .AddEntityFrameworkStores<ApplicationDbContext>();  
    services.AddRazorPages();  
}  
  
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  
{  
    if (env.IsDevelopment())  
    {  
        app.UseDeveloperExceptionPage();  
    }
```

```
+ app.UseMigrationsEndPoint();
- app.UseDatabaseErrorPage();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

For more information, see [Obsoleting DatabaseErrorPage middleware \(dotnet/aspnetcore #24987\)](#).

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](#).

Application name change

In .NET 6, [WebApplicationBuilder](#) normalizes the content root path to end with a [DirectorySeparatorChar](#). Most apps migrating from [HostBuilder](#) or [WebHostBuilder](#) won't have the same app name because they aren't normalized. For more information, see [SetApplicationName](#)

Review breaking changes

See the following resources:

- [Identity: Default Bootstrap version of UI changed](#)
- [Breaking changes for migration from version 3.1 to 5.0](#). ASP.NET Core and Entity Framework Core are also included in the list.
- [Breaking changes for migration from version 5.0 to 6.0](#): Includes ASP.NET Core and Entity Framework Core.
- [Announcements GitHub repository \(aspnet/Announcements, 6.0.0 label\)](#): Includes breaking and non-breaking information.
- [Announcements GitHub repository \(aspnet/Announcements, 5.0.0 label\)](#): Includes breaking and non-breaking information.

Migrate from ASP.NET Core 3.1 to 5.0

Article • 10/08/2024

This article explains how to update an existing ASP.NET Core 3.1 project to ASP.NET Core 5.0. For instructions on how to migrate from ASP.NET Core 3.1 to ASP.NET Core 6.0, see [Migrate from ASP.NET Core 3.1 to 6.0](#).

Prerequisites

Visual Studio

- [Visual Studio 2019 16.8 or later](#) with the **ASP.NET and web development** workload
- [.NET 5.0 SDK](#) [↗](#)

Update .NET Core SDK version in global.json

If you rely upon a [global.json](#) file to target a specific .NET Core SDK version, update the `version` property to the .NET 5.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "3.1.200"
+    "version": "5.0.100"
  }
}
```

Update the target framework

If updating a Blazor WebAssembly project, skip to the [Update Blazor WebAssembly projects](#) section. For any other ASP.NET Core project type, update the project file's **Target Framework Moniker (TFM)** to `net5.0`:

diff

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



```
<PropertyGroup>
-   <TargetFramework>netcoreapp3.1</TargetFramework>
+   <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>

</Project>
```

Delete `bin` and `obj` folders

You may need to delete the `bin` and `obj` folders. Run `dotnet nuget locals --clear all` to clear the NuGet package cache.

Changes to Blazor app routing logic in 5.0.1 and further 5.x releases up to 6.0

The computation of route precedence changed in the ASP.NET Core 5.0.1 patch release. This might affect you if you've defined catch-all routes or routes with optional parameters.

Old behavior

With the prior behavior in ASP.NET Core 5.0.0 or earlier, routes with lower precedence, such as `{*slug}`, are matched before routes with higher precedence, such as `/customer/{id}`.

New behavior

The new behavior in ASP.NET Core 5.0.1 or later more closely matches the routing behavior defined in ASP.NET Core apps, where the framework computes and establishes the route precedence for each segment first and only uses the length of the route to break ties as a secondary criteria.

Reason for change

The original behavior is considered a bug in the implementation because our goal is for the Blazor routing system to behave in the same way as the ASP.NET Core routing system for the subset of features supported by Blazor routing.

Recommended action

Add the `PreferExactMatches` attribute to the `Router` component in the `App.razor` file to opt into the correct behavior:

razor

```
<Router AppAssembly="@typeof(Program).Assembly" PreferExactMatches="@true">
```

When `PreferExactMatches` is set to `@true`, route matching prefers exact matches over wildcards.

Important

All apps should explicitly set `PreferExactMatches` to `@true`.

The ability to set `PreferExactMatches` to `@false` or leave it unset *is only provided for backward compatibility*.

When .NET 6 is released, the router will always prefer exact matches, and the `PreferExactMatches` option won't be available.

Update Blazor WebAssembly and Blazor Server projects

The guidance in this section applies to both Blazor hosting models. Sections following this section provide additional guidance specific to hosting models and app types. Apply the guidance from all relevant sections to your app.

1. In `wwwroot/index.html` of a Blazor WebAssembly app or the `Pages/_Host.cshtml` of a Blazor Server app, add a `<link>` element to the `<head>` element for styles. In the following `<link>` element `href` attribute values, the placeholder `{ASSEMBLY NAME}` is the app's assembly name.

diff

```
+<link href="{ASSEMBLY NAME}.styles.css" rel="stylesheet" />
```

Standalone Blazor WebAssembly or Blazor Server example:

diff

```
+<link href="BlazorSample.styles.css" rel="stylesheet" />
```

`Client` project of a hosted Blazor WebAssembly solution example:

diff

```
+<link href="BlazorSample.Client.styles.css" rel="stylesheet" />
```

2. Include a new namespace in the app's `_Imports.razor` file for [component virtualization](#), `Microsoft.AspNetCore.Components.Web.Virtualization`. The following `_Imports.razor` files show the default namespaces in apps generated from the Blazor project templates. The placeholder `{ASSEMBLY NAME}` is the app's assembly name.

Blazor WebAssembly (`_Imports.razor`):

razor

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {ASSEMBLY NAME}
@using {ASSEMBLY NAME}.Shared
```

Blazor Server (`_Imports.razor`):

razor

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.JSInterop
@using {ASSEMBLY NAME}
@using {ASSEMBLY NAME}.Shared
```

3. In the `MainLayout` component (`Shared/MainLayout.razor`), surround the component's HTML markup with a `<div>` element that has a `class` attribute set to `page`:

razor

```
<div class="page">

    ...

</div>
```

4. Add the following files to the `Shared` folder:

`MainLayout.razor.css`:

CSS

```
.page {
    position: relative;
    display: flex;
    flex-direction: column;
}

.main {
    flex: 1;
}

.sidebar {
    background-image: linear-gradient(180deg, rgb(5, 39, 103) 0%,
    #3a0647 70%);
}

.top-row {
    background-color: #f7f7f7;
    border-bottom: 1px solid #d6d5d5;
    justify-content: flex-end;
    height: 3.5rem;
    display: flex;
    align-items: center;
}

.top-row ::deep a, .top-row .btn-link {
    white-space: nowrap;
    margin-left: 1.5rem;
}

.top-row a:first-child {
    overflow: hidden;
    text-overflow: ellipsis;
}
```

```

@media (max-width: 767.98px) {
  .top-row:not(.auth) {
    display: none;
  }

  .top-row.auth {
    justify-content: space-between;
  }

  .top-row a, .top-row .btn-link {
    margin-left: 0;
  }
}

@media (min-width: 768px) {
  .page {
    flex-direction: row;
  }

  .sidebar {
    width: 250px;
    height: 100vh;
    position: sticky;
    top: 0;
  }

  .top-row {
    position: sticky;
    top: 0;
    z-index: 1;
  }

  .main > div {
    padding-left: 2rem !important;
    padding-right: 1.5rem !important;
  }
}

```

NavMenu.razor.css:

```

CSS

.navbar-toggler {
  background-color: rgba(255, 255, 255, 0.1);
}

.top-row {
  height: 3.5rem;
  background-color: rgba(0,0,0,0.4);
}

.navbar-brand {

```

```

    font-size: 1.1rem;
}

.oj {
    width: 2rem;
    font-size: 1.1rem;
    vertical-align: text-top;
    top: -2px;
}

.nav-item {
    font-size: 0.9rem;
    padding-bottom: 0.5rem;
}

.nav-item:first-of-type {
    padding-top: 1rem;
}

.nav-item:last-of-type {
    padding-bottom: 1rem;
}

.nav-item ::deep a {
    color: #d7d7d7;
    border-radius: 4px;
    height: 3rem;
    display: flex;
    align-items: center;
    line-height: 3rem;
}

.nav-item ::deep a.active {
    background-color: rgba(255,255,255,0.25);
    color: white;
}

.nav-item ::deep a:hover {
    background-color: rgba(255,255,255,0.1);
    color: white;
}

@media (min-width: 768px) {
    .navbar-toggler {
        display: none;
    }

    .collapse {
        /* Never collapse the sidebar for wide screens */
        display: block;
    }
}

```

5. The latest base `wwwroot/css/app.css` file of a Blazor WebAssembly app or `wwwroot/css/site.css` file of a Blazor Server app includes the following styles. Remove extra styles leaving the following styles and any that you've added to the app.

The following stylesheet only includes base styles and does **not** include custom styles added by the developer:

CSS

```
html, body {
    font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}

a, .btn-link {
    color: #0366d6;
}

.btn-primary {
    color: #fff;
    background-color: #1b6ec2;
    border-color: #1861ac;
}

.content {
    padding-top: 1.1rem;
}

.valid.modified:not([type=checkbox]) {
    outline: 1px solid #26b050;
}

.invalid {
    outline: 1px solid red;
}

.validation-message {
    color: red;
}

#blazor-error-ui {
    background: lightyellow;
    bottom: 0;
    box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
    display: none;
    left: 0;
    padding: 0.6rem 1.25rem 0.7rem 1.25rem;
    position: fixed;
    width: 100%;
    z-index: 1000;
}
```

```
#blazor-error-ui .dismiss {  
    cursor: pointer;  
    position: absolute;  
    right: 0.75rem;  
    top: 0.5rem;  
}
```

⚠ Note

The preceding example doesn't show the `@import` directive for Open Iconic icons (`open-iconic-bootstrap.css`), provided by the Blazor project template. [Open Iconic](#) [↗] was abandoned by its maintainers.

Update Blazor WebAssembly projects

Follow the guidance in the preceding [Update Blazor WebAssembly and Blazor Server projects](#) section.

For a Blazor WebAssembly project, including the `Client` project of a hosted Blazor solution, apply the following changes to the project file:

1. Update the SDK from `Microsoft.NET.Sdk.Web` to `Microsoft.NET.Sdk.BlazorWebAssembly`:

diff

```
- <Project Sdk="Microsoft.NET.Sdk.Web">  
+ <Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">
```

⚠ Note

This update only applies to standalone Blazor WebAssembly projects and the `Client` projects of hosted Blazor solutions.

2. Update the following properties:

diff

```
<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">  
  
  <PropertyGroup>  
    - <TargetFramework>netstandard2.1</TargetFramework>  
    - <RazorLangVersion>3.0</RazorLangVersion>
```



```
+    <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
```

3. Remove the package reference to

[Microsoft.AspNetCore.Components.WebAssembly.Build](#) [↗]:

```
diff

<ItemGroup>
-    <PackageReference
Include="Microsoft.AspNetCore.Components.WebAssembly.Build"
Version="3.2.1" PrivateAssets="all" />
```

4. Update other packages to their latest versions. The latest versions can be found at [NuGet.org](#) [↗].

5. In `wwwroot/index.html`, change the element that loads the `App` component to a `<div>` element with an `id` set to `app`:

```
diff

-<app>Loading...</app>
+<div id="app">Loading...</div>
```

6. In `Program.Main` (`Program.cs`), change the reference to the `<app>` element to a CSS selector by adding a hash `#` to it:

```
diff

-builder.RootComponents.Add<App>("app");
+builder.RootComponents.Add<App>("#app");
```

7. In `Program.Main` (`Program.cs`), change a default transient `HttpClient` registration to scoped, if present:

```
diff

-builder.Services.AddTransient(sp => new HttpClient
-    { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });
+builder.Services.AddScoped(sp => new HttpClient
+    { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });
```

8. In `Program.Main` (`Program.cs`) of the `Client` app of hosted Blazor solutions:

- Optionally, substitute `builder.HostEnvironment.BaseAddress` for string client base addresses.
- Change any named transient client factory registrations to scoped.

diff

```
-builder.Services.AddHttpClient("{APP_NAMESPACE}.ServerAPI",
-    client => client.BaseAddress = new Uri("https://localhost:5001"))
-    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();
-builder.Services.AddTransient(sp =>
sp.GetRequiredService<IHttpClientFactory>()
-    .CreateClient("{APP_NAMESPACE}.ServerAPI"));
+builder.Services.AddHttpClient("{APP_NAMESPACE}.ServerAPI",
+    client => client.BaseAddress = new
Uri(builder.HostEnvironment.BaseAddress))
+    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();
+builder.Services.AddScoped(sp =>
sp.GetRequiredService<IHttpClientFactory>()
+    .CreateClient("{APP_NAMESPACE}.ServerAPI"));
```

In the preceding code, the `{APP_NAMESPACE}` placeholder is the app's namespace.

Standalone Blazor WebAssembly app with Microsoft Accounts

Follow the guidance in the preceding [Update Blazor WebAssembly and Blazor Server projects](#) and [Update Blazor WebAssembly projects](#) sections.

For a standalone Blazor WebAssembly app registered in the Azure portal to use Microsoft Entra ID (ME-ID) for Microsoft Accounts:

- The app requires the `openid` and `offline_access` scopes:

C#

```
options.ProviderOptions.DefaultAccessTokenScopes.Add("openid");
options.ProviderOptions.DefaultAccessTokenScopes.Add("offline_access");
```

- In the Azure portal app registration **Authentication** blade:
 1. Remove the **Web** platform configuration.
 2. Add a **Single-page application** platform configuration with the app's redirect URI.
 3. Disable **Implicit grant** for **Access tokens** and **ID tokens**.

For more information, see [Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Accounts](#).

Standalone Blazor WebAssembly app with Microsoft Entra ID (ME-ID)

Follow the guidance in the preceding [Update Blazor WebAssembly and Blazor Server projects](#) and [Update Blazor WebAssembly projects](#) sections.

For a standalone Blazor WebAssembly app registered in the Azure portal to use Microsoft Entra ID (ME-ID):

- The app requires the `https://graph.microsoft.com/User.Read` scope:

C#

```
options.ProviderOptions.DefaultAccessTokenScopes
    .Add("https://graph.microsoft.com/User.Read");
```

- In the Azure portal app registration **Authentication** blade:
 1. Remove the **Web** platform configuration.
 2. Add a **Single-page application** platform configuration with the app's redirect URI.
 3. Disable **Implicit grant** for **Access tokens** and **ID tokens**.

For more information, see [Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID](#).

Standalone Blazor WebAssembly app with Azure Active Directory (AAD) B2C

Follow the guidance in the preceding [Update Blazor WebAssembly and Blazor Server projects](#) and [Update Blazor WebAssembly projects](#) sections.

For a standalone Blazor WebAssembly app registered in the Azure portal to use Azure Active Directory (AAD) B2C:

- The app requires the `openid` and `offline_access` scopes:

C#

```
options.ProviderOptions.DefaultAccessTokenScopes.Add("openid");
```

```
options.ProviderOptions.DefaultAccessTokenScopes.Add("offline_access");
```

- In the Azure portal app registration **Authentication** blade:
 1. Remove the **Web** platform configuration.
 2. Add a **Single-page application** platform configuration with the app's redirect URI.
 3. Disable **Implicit grant** for **Access tokens** and **ID tokens**.

For more information, see [Secure an ASP.NET Core Blazor WebAssembly standalone app with Azure Active Directory B2C](#).

Hosted Blazor WebAssembly app with Microsoft Entra ID (ME-ID) or AAD B2C

Follow the guidance in the preceding [Update Blazor WebAssembly and Blazor Server projects](#) and [Update Blazor WebAssembly projects](#) sections.

The `Client` app registration of a hosted Blazor solution that uses AAD or AAD B2C for user authentication should use a **Single-page application** Azure Apps platform configuration.

In the Azure portal `Client` app registration **Authentication** blade:

1. Remove the **Web** platform configuration.
2. Add a **Single-page application** platform configuration with the app's redirect URI.
3. Disable **Implicit grant** for **Access tokens** and **ID tokens**.

For more information, see:

- [Secure a hosted ASP.NET Core Blazor WebAssembly app with Microsoft Entra ID](#)
- [Secure a hosted ASP.NET Core Blazor WebAssembly app with Azure Active Directory B2C](#)

Update the Server project of a hosted Blazor solution

Follow the guidance in the preceding sections:

- [Update Blazor WebAssembly and Blazor Server projects](#)
- [Update Blazor WebAssembly projects](#)
- The section that applies to the app's provider with Azure Active Directory:
 - [Standalone Blazor WebAssembly app with Microsoft Accounts](#)
 - [Standalone Blazor WebAssembly app with Microsoft Entra ID \(ME-ID\)](#)

- [Standalone Blazor WebAssembly app with Azure Active Directory \(AAD\) B2C](#)

Update the `Server` project of a hosted Blazor solution as an ASP.NET Core app following the general guidance in this article.

Additionally, `Server` projects that authenticate users to client Blazor WebAssembly apps with Microsoft Entra ID (ME-ID) or B2C should adopt new Microsoft Identity v2.0 packages:

For AAD:

diff

```
-<PackageReference Include="Microsoft.AspNetCore.Authentication.AzureAD.UI"
Version="..." />
+<PackageReference Include="Microsoft.Identity.Web" Version="{VERSION}" />
+<PackageReference Include="Microsoft.Identity.Web.UI" Version="{VERSION}"
/>
```

For AAD B2C:

diff

```
-<PackageReference
Include="Microsoft.AspNetCore.Authentication.AzureADB2C.UI" Version="..." />
+<PackageReference Include="Microsoft.Identity.Web" Version="{VERSION}" />
+<PackageReference Include="Microsoft.Identity.Web.UI" Version="{VERSION}"
/>
```

For the preceding package references, determine the package versions for the `{VERSION}` placeholders at NuGet.org:

- [Microsoft.Identity.Web](#) ↗
- [Microsoft.Identity.Web.UI](#) ↗

📌 Note

The SDK of the `Server` project in a hosted Blazor WebAssembly solution remains `Microsoft.NET.Sdk.Web`:

XML

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

For more information, see:

- [Secure a hosted ASP.NET Core Blazor WebAssembly app with Microsoft Entra ID](#)
- [Secure a hosted ASP.NET Core Blazor WebAssembly app with Azure Active Directory B2C](#)

Clean and rebuild the solution

After migrating the app or solution to .NET 5, clean and rebuild the app or solution. If package incompatibilities exist between new package references and cached packages:

1. Clear NuGet package caches by executing the following [dotnet nuget locals](#) command in a command shell:

```
.NET CLI  
  
dotnet nuget locals --clear all
```

2. Clean and rebuild the app or solution.

Troubleshoot

Follow the *Troubleshoot* guidance at the end of the Blazor WebAssembly security topic that applies to your app:

Standalone Blazor WebAssembly apps:

- [General guidance for OIDC providers and the WebAssembly Authentication Library](#)
- [Microsoft Accounts](#)
- [Microsoft Entra ID \(ME-ID\)](#)
- [Azure Active Directory \(AAD\) B2C](#)

Hosted Blazor WebAssembly apps:

- [Microsoft Entra ID \(ME-ID\)](#)
- [Azure Active Directory \(AAD\) B2C](#)
- [Identity Server](#)

Unauthorized client for Microsoft Entra ID (ME-ID)

After upgrading a Blazor WebAssembly app that uses AAD for authentication, you may receive the following error on the login callback to the app after the user signs in with AAD:

info: Microsoft.AspNetCore.Authorization.DefaultAuthorizationService[2]

Authorization failed. These requirements were not met:

DenyAnonymousAuthorizationRequirement: Requires an authenticated user.

Login callback error from AAD:

- Error: `unauthorized_client`
- Description: `AADB2C90058: The provided application is not configured to allow public clients.`

To resolve the error:

1. In the Azure portal, access the [app's manifest](#).
2. Set the `allowPublicClient` attribute to `null` or `true`.

Update a Blazor Progressive Web Application (PWA)

Add the following item to the PWA app's project file:

XML

```
<ItemGroup>
  <ServiceWorker Include="wwwroot\service-worker.js"
    PublishedContent="wwwroot\service-worker.published.js" />
</ItemGroup>
```

Remove preview CSS isolation stylesheet link

If the project's `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server) contains a stylesheet `<link>` element for `scoped.styles.css` from an earlier 5.0 preview release, remove the `<link>` tag:

diff

```
-<link href="_framework/scoped.styles.css/" rel="stylesheet" />
```

Update Razor class libraries (RCLs)

Migrate Razor class libraries (RCLs) to take advantage of new APIs or features that are introduced as part of ASP.NET Core 5.0.

To update a RCL that targets components:

1. Update the following properties in the project file:

```
diff

<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
-    <TargetFramework>netstandard2.0</TargetFramework>
-    <RazorLangVersion>3.0</RazorLangVersion>
+    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
```

2. Update other packages to their latest versions. The latest versions can be found at [NuGet.org](https://www.nuget.org).

To update an RCL targeting MVC, update the following properties in the project file:

```
diff

<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
-    <TargetFramework>netcoreapp3.1</TargetFramework>
+    <TargetFramework>net5.0</TargetFramework>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>
```

Update package references

In the project file, update each [Microsoft.AspNetCore.*](#), [Microsoft.EntityFrameworkCore.*](#), [Microsoft.Extensions.*](#), and [System.Net.Http.Json](#) package reference's `Version` attribute to 5.0.0 or later. For example:

```
diff

<ItemGroup>
-    <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="3.1.6" />
-    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="3.1.6">
-    <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
```



```
Version="3.1.6" />
- <PackageReference Include="System.Net.Http.Json" Version="3.2.1" />
+ <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="5.0.0" />
+ <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="5.0.0">
+ <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="5.0.0" />
+ <PackageReference Include="System.Net.Http.Json" Version="5.0.0" />
</ItemGroup>
```

Update Docker images

For apps using Docker, update your *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 5.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 3.1 and 5.0:

diff

```
- docker pull mcr.microsoft.com/dotnet/core/aspnet:3.1
+ docker pull mcr.microsoft.com/dotnet/aspnet:5.0
```

As part of the move to ".NET" as the product name, the Docker images moved from the `mcr.microsoft.com/dotnet/core` repositories to `mcr.microsoft.com/dotnet`. For more information, see [dotnet/dotnet-docker#1939](https://github.com/dotnet/dotnet-docker#1939).

Model binding changes in ASP.NET Core MVC and Razor Pages

DateTime values are model bound as UTC times

In ASP.NET Core 3.1 and earlier, `DateTime` values were model-bound as local time, where the timezone was determined by the server. `DateTime` values bound from input formatting (JSON) and `DateTimeOffset` values were bound as UTC timezones.

In ASP.NET Core 5.0 and later, model binding consistently binds `DateTime` values with the UTC timezone.

To retain the previous behavior, remove the `DateTimeModelBinderProvider` in `Startup.ConfigureServices`:

C#

```
services.AddControllersWithViews(options =>
    options.ModelBinderProviders.RemoveType<DateTimeModelBinderProvider>());
```

ComplexObjectModelBinderProvider \ ComplexObjectModelBinder replace ComplexTypeModelBinderProvider \ ComplexTypeModelBinder

To add support for model binding [C# 9 record types](#), the [ComplexTypeModelBinderProvider](#) is:

- Annotated as obsolete.
- No longer registered by default.

Apps that rely on the presence of the `ComplexTypeModelBinderProvider` in the `ModelBinderProviders` collection need to reference the new binder provider:

diff

```
- var complexModelBinderProvider =
options.ModelBinderProviders.OfType<ComplexTypeModelBinderProvider>();
+ var complexModelBinderProvider =
options.ModelBinderProviders.OfType<ComplexObjectModelBinderProvider>();
```

UseDatabaseErrorPage obsolete

The ASP.NET Core 3.1 templates that include an option for individual user accounts generate a call to [UseDatabaseErrorPage](#). `UseDatabaseErrorPage` is now obsolete and should be replaced with a combination of `AddDatabaseDeveloperPageExceptionFilter` and `UseMigrationsEndPoint`, as shown in the following code:

diff

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
+   services.AddDatabaseDeveloperPageExceptionFilter();
    services.AddDefaultIdentity<IdentityUser>(options =>
        options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();
```

```

}

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

```

For more information, see [this GitHub issue](#).

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](#).

Package reference changes affecting some NuGet packages

With the migration of some `Microsoft.Extensions.*` NuGet packages from the [dotnet/extensions](#) repository to [dotnet/runtime](#), as described in [Migrating dotnet/extensions content to dotnet/runtime and dotnet/aspnetcore \(aspnet/Announcements #411\)](#), packaging changes are being applied to some of the migrated packages. These changes often result in namespace changes for .NET API.

To research APIs further for app namespace changes when migrating to 5.0, use the [.NET API browser](#).



Migrate Microsoft.Identity.Web

The following wiki pages explain how to migrate Microsoft.Identity.Web from ASP.NET Core 3.1 to 5.0:

- [Microsoft.Identity.Web in web apps](#)

- [Microsoft.Identity.Web in web APIs](#) 

The following tutorials also explain the migration:

- [An ASP.NET Core Web app signing-in users with the Microsoft identity platform in your organization](#) . See **Option 2: Create the sample from the command line**.
- [Sign-in a user with the Microsoft identity platform in a WPF Desktop application and call an ASP.NET Core Web API](#) . See **How was the code created**.

Review breaking changes

For breaking changes from .NET Core 3.1 to .NET 5.0, see [Breaking changes for migration from version 3.1 to 5.0](#). ASP.NET Core and Entity Framework Core are also included in the list.

Migrate from ASP.NET Core 3.0 to 3.1


Article • 06/18/2024

By [Scott Addie](#) 

This article explains how to update an existing ASP.NET Core 3.0 project to ASP.NET Core 3.1.

Prerequisites

Visual Studio

- [Visual Studio 2019 16.4 or later](#) with the **ASP.NET and web development** workload
- [.NET Core 3.1 SDK](#) 

Update .NET Core SDK version in global.json

If you rely upon a [global.json](#) file to target a specific .NET Core SDK version, update the `version` property to the 3.1 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "3.0.101"
+    "version": "3.1.101"
  }
}
```

Update the target framework

In the project file, update the [Target Framework Moniker \(TFM\)](#) to `netcoreapp3.1`:

diff

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

  <PropertyGroup>
-    <TargetFramework>netcoreapp3.0</TargetFramework>
```

```
+    <TargetFramework>netcoreapp3.1</TargetFramework>
    </PropertyGroup>

</Project>
```

Update package references

In the project file, update each `Microsoft.AspNetCore.*` package reference's `Version` attribute to 3.1.0 or later. For example:

diff

```
<ItemGroup>
-    <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson"
Version="3.0.0" />
-    <PackageReference
Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="3.0.0"
Condition="'$(Configuration)' == 'Debug'" />
+    <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson"
Version="3.1.1" />
+    <PackageReference
Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="3.1.1"
Condition="'$(Configuration)' == 'Debug'" />
</ItemGroup>
```

Update Docker images

For apps using Docker, use a base image that includes ASP.NET Core 3.1. For example:

Console

```
docker pull mcr.microsoft.com/dotnet/aspnet:3.1
```

React to SameSite cookie changes

The `SameSite` attribute implementations for HTTP cookies changed between ASP.NET Core 3.0 and 3.1. For actions to be taken, see the following resources:

- [Work with SameSite cookies in ASP.NET Core](#)
- [aspnet/Announcements#390](#) ↗
- [Upcoming SameSite cookie changes in ASP.NET and ASP.NET Core](#) ↗

Publish with Visual Studio

In the `.pubxml` file update the `TargetFramework` to 3.1:

XML

```
- <TargetFramework>netcoreapp3.0</TargetFramework>
+ <TargetFramework>netcoreapp3.1</TargetFramework>
```

Review breaking changes

Review 3.0-to-3.1 breaking changes across .NET Core, ASP.NET Core, and Entity Framework Core at [Breaking changes for migration from version 3.0 to 3.1](#).

Optional changes

The following changes are optional.

Use the Component Tag Helper

ASP.NET Core 3.1 introduces a `Component` Tag Helper. The Tag Helper can replace the `RenderComponentAsync<TComponent>` HTML helper method in a Blazor project. For example:

diff

```
- @(await Html.RenderComponentAsync<Counter>(RenderMode.ServerPrerendered,
new { IncrementAmount = 10 })))
+ <component type="typeof(Counter)" render-mode="ServerPrerendered" param-
IncrementAmount="10" />
```

For more information, see [Prerender and integrate ASP.NET Core Razor components](#).

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](#).

Migrate from ASP.NET Core 2.2 to 3.0

Article • 08/30/2024

By [Scott Addie](#) and [Rick Anderson](#)

This article explains how to update an existing ASP.NET Core 2.2 project to ASP.NET Core 3.0. It might be helpful to create a new ASP.NET Core 3.0 project to:

- Compare with the ASP.NET Core 2.2 code.
- Copy the relevant changes to your ASP.NET Core 3.0 project.

Prerequisites

Visual Studio

- [Visual Studio 2019](#) with the **ASP.NET and web development** workload
- [.NET Core 3.0 SDK](#)

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 3.0 version installed on your machine:

JSON

```
{
  "sdk": {
    "version": "3.0.100"
  }
}
```

Update the project file

Update the Target Framework

ASP.NET Core 3.0 and later only run on .NET Core. Set the [Target Framework Moniker \(TFM\)](#) to `netcoreapp3.0`:

XML

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Remove obsolete package references

A large number of NuGet packages aren't produced for ASP.NET Core 3.0. Such package references should be removed from your project file. Consider the following project file for an ASP.NET Core 2.2 web app:

XML

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design"
Version="2.2.0" PrivateAssets="All" />
  </ItemGroup>

</Project>
```

The updated project file for ASP.NET Core 3.0:

XML

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

The updated ASP.NET Core 3.0 project file:

- In the `<PropertyGroup>`:

- Updates the TFM to `netcoreapp3.0`
- Removes the `<AspNetCoreHostingModel>` element. For more information, see [In-process hosting model](#) in this document.
- In the `<ItemGroup>`:
 - `Microsoft.AspNetCore.App` is removed. For more information, see [Framework reference](#) in this document.
 - `Microsoft.AspNetCore.Razor.Design` is removed and in the following list of packages no longer being produced.

To see the full list of packages that are no longer produced, select the following expand list:

▼ Click to expand the list of packages no longer being produced

- Microsoft.AspNetCore
- Microsoft.AspNetCore.All
- Microsoft.AspNetCore.App
- Microsoft.AspNetCore.Antiforgery
- Microsoft.AspNetCore.Authentication
- Microsoft.AspNetCore.Authentication.Abstractions
- Microsoft.AspNetCore.Authentication.Cookies
- Microsoft.AspNetCore.Authentication.Core
- Microsoft.AspNetCore.Authentication.OAuth
- Microsoft.AspNetCore.Authorization.Policy
- Microsoft.AspNetCore.CookiePolicy
- Microsoft.AspNetCore.Cors
- Microsoft.AspNetCore.Diagnostics
- Microsoft.AspNetCore.Diagnostics.HealthChecks
- Microsoft.AspNetCore.HostFiltering
- Microsoft.AspNetCore.Hosting
- Microsoft.AspNetCore.Hosting.Abstractions
- Microsoft.AspNetCore.Hosting.Server.Abstractions
- Microsoft.AspNetCore.Http
- Microsoft.AspNetCore.Http.Abstractions
- Microsoft.AspNetCore.Http.Connections
- Microsoft.AspNetCore.Http.Extensions
- Microsoft.AspNetCore.HttpOverrides
- Microsoft.AspNetCore.HttpsPolicy
- Microsoft.AspNetCore.Identity
- Microsoft.AspNetCore.Localization
- Microsoft.AspNetCore.Localization.Routing

- Microsoft.AspNetCore.Mvc
- Microsoft.AspNetCore.Mvc.Abstractions
- Microsoft.AspNetCore.Mvc.Analyzers
- Microsoft.AspNetCore.Mvc.ApiExplorer
- Microsoft.AspNetCore.Mvc.Api.Analyzers
- Microsoft.AspNetCore.Mvc.Core
- Microsoft.AspNetCore.Mvc.Cors
- Microsoft.AspNetCore.Mvc.DataAnnotations
- Microsoft.AspNetCore.Mvc.Formatters.Json
- Microsoft.AspNetCore.Mvc.Formatters.Xml
- Microsoft.AspNetCore.Mvc.Localization
- Microsoft.AspNetCore.Mvc.Razor
- Microsoft.AspNetCore.Mvc.Razor.ViewCompilation
- Microsoft.AspNetCore.Mvc.RazorPages
- Microsoft.AspNetCore.Mvc.TagHelpers
- Microsoft.AspNetCore.Mvc.ViewFeatures
- Microsoft.AspNetCore.Razor
- Microsoft.AspNetCore.Razor.Runtime
- Microsoft.AspNetCore.Razor.Design
- Microsoft.AspNetCore.ResponseCaching
- Microsoft.AspNetCore.ResponseCaching.Abstractions
- Microsoft.AspNetCore.ResponseCompression
- Microsoft.AspNetCore.Rewrite
- Microsoft.AspNetCore.Routing
- Microsoft.AspNetCore.Routing.Abstractions
- Microsoft.AspNetCore.Server.HttpSys
- Microsoft.AspNetCore.Server.IIS
- Microsoft.AspNetCore.Server.IISIntegration
- Microsoft.AspNetCore.Server.Kestrel
- Microsoft.AspNetCore.Server.Kestrel.Core
- Microsoft.AspNetCore.Server.Kestrel.Https
- Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions
- Microsoft.AspNetCore.Server.Kestrel.Transport.Sockets
- Microsoft.AspNetCore.Session
- Microsoft.AspNetCore.SignalR
- Microsoft.AspNetCore.SignalR.Core
- Microsoft.AspNetCore.StaticFiles
- Microsoft.AspNetCore.WebSockets
- Microsoft.AspNetCore.WebUtilities
- Microsoft.Net.Http.Headers

Review breaking changes

[Review breaking changes](#)

Framework reference

Features of ASP.NET Core that were available through one of the packages listed above are available as part of the `Microsoft.AspNetCore.App` shared framework. The *shared framework* is the set of assemblies (.dll files) that are installed on the machine and includes a runtime component and a targeting pack. For more information, see [The shared framework](#).

- Projects that target the `Microsoft.NET.Sdk.Web` SDK implicitly reference the `Microsoft.AspNetCore.App` framework.

No additional references are required for these projects:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>
  ...
</Project>
```

- Projects that target `Microsoft.NET.Sdk` or `Microsoft.NET.Sdk.Razor` SDK, should add an explicit `FrameworkReference` to `Microsoft.AspNetCore.App`:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
  ...
</Project>
```

Framework-dependent builds using Docker

Framework-dependent builds of console apps that use a package that depends on the ASP.NET Core [shared framework](#) may give the following runtime error:

Console

```
It was not possible to find any compatible framework version
The specified framework 'Microsoft.AspNetCore.App', version '3.0.0' was not
found.
- No frameworks were found.
```

`Microsoft.AspNetCore.App` is the shared framework containing the ASP.NET Core runtime and is only present on the `dotnet/core/aspnet` Docker image. The 3.0 SDK reduces the size of framework-dependent builds using ASP.NET Core by not including duplicate copies of libraries that are available in the shared framework. This is a potential savings of up to 18 MB, but it requires that the ASP.NET Core runtime be present / installed to run the app.

To determine if the app has a dependency (either direct or indirect) on the ASP.NET Core shared framework, examine the `runtimeconfig.json` file generated during a build/publish of your app. The following JSON file shows a dependency on the ASP.NET Core shared framework:

JSON

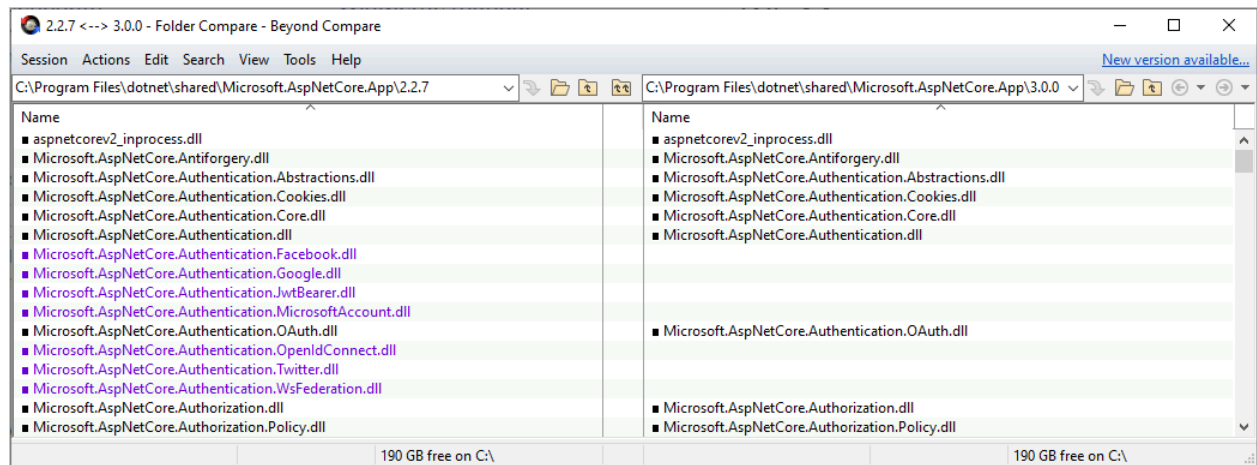
```
{
  "runtimeOptions": {
    "tfm": "netcoreapp3.0",
    "framework": {
      "name": "Microsoft.AspNetCore.App",
      "version": "3.0.0"
    },
    "configProperties": {
      "System.GC.Server": true
    }
  }
}
```

If your app is using Docker, use a base image that includes ASP.NET Core 3.0. For example, `docker pull mcr.microsoft.com/dotnet/core/aspnet:3.0`.

Add package references for removed assemblies

ASP.NET Core 3.0 removes some assemblies that were previously part of the `Microsoft.AspNetCore.App` package reference. To visualize which assemblies were

removed, compare the two shared framework folders. For example, a comparison of versions 2.2.7 and 3.0.0:



To continue using features provided by the removed assemblies, reference the 3.0 versions of the corresponding packages:

- A template-generated web app with **Individual User Accounts** requires adding the following packages:

XML

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UserSecretsId>My-secret</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
      Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore"
      Version="3.0.0" />
    <PackageReference
      Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore"
      Version="3.0.0" />
    <PackageReference Include="Microsoft.AspNetCore.Identity.UI"
      Version="3.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
      Version="3.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
      Version="3.0.0" />
  </ItemGroup>

</Project>
```

- [Microsoft.EntityFrameworkCore](#)

For more information on referencing the database provider-specific package, see [Database Providers](#).

- Identity UI

Support for [Identity UI](#) can be added by referencing the [Microsoft.AspNetCore.Identity.UI](#) package.

- SPA Services

- [Microsoft.AspNetCore.SpaServices](#)
- [Microsoft.AspNetCore.SpaServices.Extensions](#)

- Authentication: Support for third-party authentication flows are available as NuGet packages:

- Facebook OAuth ([Microsoft.AspNetCore.Authentication.Facebook](#))
- Google OAuth ([Microsoft.AspNetCore.Authentication.Google](#))
- Microsoft Account authentication ([Microsoft.AspNetCore.Authentication.MicrosoftAccount](#))
- OpenID Connect authentication ([Microsoft.AspNetCore.Authentication.OpenIdConnect](#))
- OpenID Connect bearer token ([Microsoft.AspNetCore.Authentication.JwtBearer](#))
- Twitter OAuth ([Microsoft.AspNetCore.Authentication.Twitter](#))
- WsFederation authentication ([Microsoft.AspNetCore.Authentication.WsFederation](#))

- Formatting and content negotiation support for `System.Net.HttpClient`: The [Microsoft.AspNetCore.WebApi.Client](#) NuGet package provides useful extensibility to `System.Net.HttpClient` with APIs such as `ReadAsStringAsync` and `PostJsonAsync`. However, this package depends on `Newtonsoft.Json`, not `System.Text.Json`. That means, for example, that serialization property names specified by `JsonPropertyNameAttribute` (`System.Text.Json`) are ignored. There's a newer NuGet package that contains similar extension methods but uses `System.Text.Json`: [System.Net.Http.Json](#).

- Razor runtime compilation: Support for runtime compilation of Razor views and pages is now part of [Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation](#).

- MVC `Newtonsoft.Json` (Json.NET) support: Support for using MVC with `Newtonsoft.Json` is now part of [Microsoft.AspNetCore.Mvc.NewtonsoftJson](#).

Startup changes

The following image shows the deleted and changed lines in an ASP.NET Core 2.2 Razor Pages Web app:

```
Diff - Startup.cs;HEAD vs. Startup.cs | Web1.csproj | Startup.cs | Web1
Startup.cs;HEAD
[C#] Miscellaneous Files | Web1.Startup | Startup(IConfiguration configurat
1  using Microsoft.AspNetCore.Builder;
2  using Microsoft.AspNetCore.Hosting;
3  using Microsoft.AspNetCore.Mvc;
4  using Microsoft.Extensions.Configuration;
5  using Microsoft.Extensions.DependencyInjection;
6  //
7  namespace Web1
8  {
9      public class Startup
10     {
11         public Startup(IConfiguration configuration)
12         {
13             Configuration = configuration;
14         }
15
16         public IConfiguration Configuration { get; }
17
18         public void ConfigureServices(IServiceCollection services)
19         {
20             services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
21         }
22
23         public void Configure(IApplicationBuilder app, IHostingEnvironment env)
24         {
25             if (env.IsDevelopment())
26             {
27                 app.UseDeveloperExceptionPage();
28             }
29             else
30             {
31                 app.UseExceptionHandler("/Error");
32                 app.UseHsts();
33             }
34
35             app.UseHttpsRedirection();
36             app.UseStaticFiles();
37
38             app.UseMvc();
39         }
40     }
41 }
```

In the preceding image, deleted code is shown in red. The deleted code doesn't show cookie options code, which was deleted prior to comparing the files.

The following image shows the added and changed lines in an ASP.NET Core 3.0 Razor Pages Web app:


```

Startup.cs
Web1 Web1.Startup Startup(IConfiguration con
1 using Microsoft.AspNetCore.Builder;
2 using Microsoft.AspNetCore.Hosting;
3 ///////////////////////////////////////////////////
4 using Microsoft.Extensions.Configuration;
5 using Microsoft.Extensions.DependencyInjection;
6 using Microsoft.Extensions.Hosting;
7 namespace Web1
8 {
9     public class Startup
10    {
11        public Startup(IConfiguration configuration)
12        {
13            Configuration = configuration;
14        }
15
16        public IConfiguration Configuration { get; }
17
18        public void ConfigureServices(IServiceCollection services)
19        {
20            services.AddRazorPages();
21        }
22
23        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
24        {
25            if (env.IsDevelopment())
26            {
27                app.UseDeveloperExceptionPage();
28            }
29            else
30            {
31                app.UseExceptionHandler("/Error");
32                app.UseHsts();
33            }
34
35            app.UseHttpsRedirection();
36            app.UseStaticFiles();
37
38            app.UseRouting();
39
40            app.UseAuthorization();
41
42            app.UseEndpoints(endpoints =>
43            {
44                endpoints.MapRazorPages();
45            });
46        }
47    }
48 }
49

```

In the preceding image, added code is shown in green. For information on the following changes:

- `services.AddMvc` to `services.AddRazorPages`, see [MVC service registration](#) in this document.
- `CompatibilityVersion`, see [Compatibility version for ASP.NET Core MVC](#).
- `IHostingEnvironment` to `IWebHostEnvironment`, see [this GitHub announcement](#).
- `app.UseAuthorization` was added to the templates to show the order authorization middleware must be added. If the app doesn't use authorization, you can safely remove the call to `app.UseAuthorization`.
- `app.UseEndpoints`, see [Razor Pages](#) or [Migrate Startup.Configure](#) in this document.

Analyzer support

Projects that target `Microsoft.NET.Sdk.Web` implicitly reference analyzers previously shipped as part of the [Microsoft.AspNetCore.Mvc.Analyzers](#) package. No additional references are required to enable these.

If your app uses [API analyzers](#) previously shipped using the [Microsoft.AspNetCore.Mvc.Api.Analyzers](#) package, edit your project file to reference the analyzers shipped as part of the .NET Core Web SDK:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>
  </PropertyGroup>

  ...
</Project>
```

Razor class library

Razor class library projects that provide UI components for MVC must set the `AddRazorSupportForMvc` property in the project file:

XML

```
<PropertyGroup>
  <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
</PropertyGroup>
```

In-process hosting model

Projects default to the [in-process hosting model](#) in ASP.NET Core 3.0 or later. You may optionally remove the `<AspNetCoreHostingModel>` property in the project file if its value is `InProcess`.

Kestrel

Configuration

Migrate Kestrel configuration to the [web host builder](#) provided by

`ConfigureWebHostDefaults` (`Program.cs`):

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel(serverOptions =>
            {
                // Set properties and call methods on options
            })
            .UseStartup<Startup>();
        });
```

If the app creates the host manually with `ConfigureWebHost` instead of `ConfigureWebHostDefaults`, call `UseKestrel` on the web host builder:

C#

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

    host.Run();
}
```

Connection Middleware replaces Connection Adapters

Connection Adapters

(`Microsoft.AspNetCore.Server.Kestrel.Core.Adapter.Internal.IConnectionAdapter`) have been removed from Kestrel. Replace Connection Adapters with Connection Middleware. Connection Middleware is similar to HTTP Middleware in the ASP.NET Core pipeline but for lower-level connections. HTTPS and connection logging:

- Have been moved from Connection Adapters to Connection Middleware.
- These extension methods work as in previous versions of ASP.NET Core.

For more information, see [the TlsFilterConnectionHandler example in the ListenOptions.Protocols section of the Kestrel article](#).

Transport abstractions moved and made public

The Kestrel transport layer has been exposed as a public interface in `Connections.Abstractions`. As part of these updates:

- `Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions` and associated types have been removed.
- `NoDelay` was moved from [ListenOptions](#) to the transport options.
- `Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions.Internal.SchedulingMode` was removed from [KestrelServerOptions](#).

For more information, see the following GitHub resources:

- [Client/server networking abstractions \(dotnet/AspNetCore #10308\)](#) ↗
- [Implement new bedrock listener abstraction and re-plat Kestrel on top \(dotnet/AspNetCore #10321\)](#) ↗

Kestrel Request trailer headers

For apps that target earlier versions of ASP.NET Core:

- Kestrel adds HTTP/1.1 chunked trailer headers into the request headers collection.
- Trailers are available after the request body is read to the end.

This causes some concerns about ambiguity between headers and trailers, so the trailers have been moved to a new collection (`RequestTrailerExtensions`) in 3.0.

HTTP/2 request trailers are:

- Not available in ASP.NET Core 2.2.
- Available in 3.0 as `RequestTrailerExtensions`.

New request extension methods are present to access these trailers. As with HTTP/1.1, trailers are available after the request body is read to the end.

For the 3.0 release, the following `RequestTrailerExtensions` methods are available:

- `GetDeclaredTrailers`: Gets the request `Trailer` header that lists which trailers to expect after the body.
- `SupportsTrailers`: Indicates if the request supports receiving trailer headers.
- `CheckTrailersAvailable`: Checks if the request supports trailers and if they're available to be read. This check doesn't assume that there are trailers to read. There might be no trailers to read even if `true` is returned by this method.
- `GetTrailer`: Gets the requested trailing header from the response. Check `SupportsTrailers` before calling `GetTrailer`, or a `NotSupportedException` may occur if the request doesn't support trailing headers.

For more information, see [Put request trailers in a separate collection \(dotnet/AspNetCore #10410\)](#).

AllowSynchronousIO disabled

`AllowSynchronousIO` enables or disables synchronous I/O APIs, such as `HttpRequest.Body.Read`, `HttpResponse.Body.Write`, and `Stream.Flush`. These APIs are a source of thread starvation leading to app crashes. In 3.0, `AllowSynchronousIO` is disabled by default. For more information, see [the Synchronous I/O section in the Kestrel article](#).

If synchronous I/O is needed, it can be enabled by configuring the `AllowSynchronousIO` option on the server being used (when calling `ConfigureKestrel`, for example, if using Kestrel). Note that servers (Kestrel, HttpSys, TestServer, etc.) all have their own `AllowSynchronousIO` option that won't affect other servers. Synchronous I/O can be enabled for all servers on a per-request basis using the `IHttpBodyControlFeature.AllowSynchronousIO` option:

C#

```
var syncIOFeature = HttpContext.Features.Get<IHttpBodyControlFeature>();

if (syncIOFeature != null)
{
    syncIOFeature.AllowSynchronousIO = true;
}
```

If you have trouble with `TextWriter` implementations or other streams that call synchronous APIs in `Dispose`, call the new `DisposeAsync` API instead.

For more information, see [\[Announcement\] AllowSynchronousIO disabled in all servers \(dotnet/AspNetCore #7644\)](#).

Output formatter buffering

[Newtonsoft.Json](#), [XmlSerializer](#), and [DataContractSerializer](#) based output formatters only support synchronous serialization. To allow these formatters to work with the [AllowSynchronousIO](#) restrictions of the server, MVC buffers the output of these formatters before writing to disk. As a result of buffering, MVC will include the Content-Length header when responding using these formatters.

[System.Text.Json](#) supports asynchronous serialization and consequently the `System.Text.Json` based formatter does not buffer. Consider using this formatter for improved performance.

To disable buffering, applications can configure [SuppressOutputFormatterBuffering](#) in their startup:

C#

```
services.AddControllers(options => options.SuppressOutputFormatterBuffering  
= true)
```

Note that this may result in the application throwing a runtime exception if `AllowSynchronousIO` isn't also configured.

Microsoft.AspNetCore.Server.Kestrel.Https assembly removed

In ASP.NET Core 2.1, the contents of *Microsoft.AspNetCore.Server.Kestrel.Https.dll* were moved to *Microsoft.AspNetCore.Server.Kestrel.Core.dll*. This was a non-breaking update using `TypeForwardedTo` attributes. For 3.0, the empty *Microsoft.AspNetCore.Server.Kestrel.Https.dll* assembly and the NuGet package have been removed.

Libraries referencing [Microsoft.AspNetCore.Server.Kestrel.Https](#) should update ASP.NET Core dependencies to 2.1 or later.

Apps and libraries targeting ASP.NET Core 2.1 or later should remove any direct references to the [Microsoft.AspNetCore.Server.Kestrel.Https](#) package.

Newtonsoft.Json (Json.NET) support

As part of the work to [improve the ASP.NET Core shared framework](#), [Newtonsoft.Json \(Json.NET\)](#) has been removed from the ASP.NET Core shared framework.

The default JSON serializer for ASP.NET Core is now [System.Text.Json](#), which is new in .NET Core 3.0. Consider using `System.Text.Json` when possible. It's high-performance and doesn't require an additional library dependency. However, since `System.Text.Json` is new, it might currently be missing features that your app needs. For more information, see [How to migrate from Newtonsoft.Json to System.Text.Json](#).

Use Newtonsoft.Json in an ASP.NET Core 3.0 SignalR project

- Install the [Microsoft.AspNetCore.SignalR.Protocols.NewtonsoftJson](#) [NuGet](#) package.
- On the client, chain an `AddNewtonsoftJsonProtocol` method call to the `HubConnectionBuilder` instance:

```
C#  
  
new HubConnectionBuilder()  
    .WithUrl("/chathub")  
    .AddNewtonsoftJsonProtocol(...)  
    .Build();
```

- On the server, chain an `AddNewtonsoftJsonProtocol` method call to the `AddSignalR` method call in `Startup.ConfigureServices`:

```
C#  
  
services.AddSignalR()  
    .AddNewtonsoftJsonProtocol(...);
```

Use Newtonsoft.Json in an ASP.NET Core 3.0 MVC project

- Install the [Microsoft.AspNetCore.Mvc.NewtonsoftJson](#) [NuGet](#) package.
- Update `Startup.ConfigureServices` to call `AddNewtonsoftJson`.

```
C#  
  
services.AddMvc()  
    .AddNewtonsoftJson();
```

`AddNewtonsoftJson` is compatible with the new MVC service registration methods:

- `AddRazorPages`
- `AddControllersWithViews`
- `AddControllers`

C#

```
services.AddControllers()  
        .AddNewtonsoftJson();
```

`Newtonsoft.Json` settings can be set in the call to `AddNewtonsoftJson`:

C#

```
services.AddMvc()  
        .AddNewtonsoftJson(options =>  
            options.SerializerSettings.ContractResolver =  
                new CamelCasePropertyNamesContractResolver());
```

Note: If the `AddNewtonsoftJson` method isn't available, make sure that you installed the [Microsoft.AspNetCore.Mvc.NewtonsoftJson](#) package. A common error is to install the [Newtonsoft.Json](#) package instead of the [Microsoft.AspNetCore.Mvc.NewtonsoftJson](#) package.

For more information, see [Add Newtonsoft.Json-based JSON format support](#).

MVC service registration

ASP.NET Core 3.0 adds new options for registering MVC scenarios inside `Startup.ConfigureServices`.

Three new top-level extension methods related to MVC scenarios on `IServiceCollection` are available. Templates use these new methods instead of `AddMvc`. However, `AddMvc` continues to behave as it has in previous releases.

The following example adds support for controllers and API-related features, but not views or pages. The API template uses this code:

C#

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


The following example adds support for controllers, API-related features, and views, but not pages. The Web Application (MVC) template uses this code:

C#

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

The following example adds support for Razor Pages and minimal controller support. The Web Application template uses this code:

C#

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

The new methods can also be combined. The following example is equivalent to calling `AddMvc` in ASP.NET Core 2.2:

C#

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

Routing startup code

If an app calls `UseMvc` or `UseSignalR`, migrate the app to [Endpoint Routing](#) if possible. To improve Endpoint Routing compatibility with previous versions of MVC, we've reverted some of the changes in URL generation introduced in ASP.NET Core 2.2. If you experienced problems using Endpoint Routing in 2.2, expect improvements in ASP.NET Core 3.0 with the following exceptions:

- If the app implements `IRouter` or inherits from `Route`, use [DynamicRouteValuesTransformer](#) [↗](#) as the replacement.
- If the app directly accesses `RouteData.Routers` inside MVC to parse URLs, you can replace this with use of [LinkParser.ParsePathByEndpointName](#).
 - Define the route with a route name.

- Use `LinkParser.ParsePathByEndpointName` and pass in the desired route name.

Endpoint Routing supports the same route pattern syntax and route pattern authoring features as `IRouter`. Endpoint Routing supports `IRouteConstraint`. Endpoint routing supports `[Route]`, `[HttpGet]`, and the other MVC routing attributes.

For most applications, only `Startup` requires changes.

Migrate Startup.Configure

General advice:

- Add `UseRouting`.
- If the app calls `UseStaticFiles`, place `UseStaticFiles` **before** `UseRouting`.
- If the app uses authentication/authorization features such as `AuthorizePage` or `[Authorize]`, place the call to `UseAuthentication` and `UseAuthorization`: **after**, `UseRouting` and `UseCors`, but before `UseEndpoints`:

```
C#

public void Configure(IApplicationBuilder app)
{
    ...

    app.UseStaticFiles();

    app.UseRouting();
    app.UseCors();

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

    app.UseEndpoints(endpoints => {
        endpoints.MapControllers();
    });
}
```

- Replace `UseMvc` or `UseSignalR` with `UseEndpoints`.
- If the app uses [CORS](#) scenarios, such as `[EnableCors]`, place the call to `UseCors` before any other middleware that use CORS (for example, place `UseCors` before `UseAuthentication`, `UseAuthorization`, and `UseEndpoints`).
- Replace `IHostingEnvironment` with `IWebHostEnvironment` and add a `using` statement for the [Microsoft.AspNetCore.Hosting](#) namespace.

- Replace `IApplicationLifetime` with `IHostApplicationLifetime` (`Microsoft.Extensions.Hosting` namespace).
- Replace `EnvironmentName` with `Environments` (`Microsoft.Extensions.Hosting` namespace).

The following code is an example of `Startup.Configure` in a typical ASP.NET Core 2.2 app:

C#

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

    app.UseStaticFiles();

    app.UseAuthentication();

    app.UseSignalR(hubs =>
    {
        hubs.MapHub<ChatHub>("/chat");
    });

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

After updating the previous `Startup.Configure` code:

C#

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

    app.UseStaticFiles();

    app.UseRouting();

    app.UseCors();

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

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/chat");
    });
}
```

```
endpoints.MapControllerRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
}
```

Warning

For most apps, calls to `UseAuthentication`, `UseAuthorization`, and `UseCors` must appear between the calls to `UseRouting` and `UseEndpoints` to be effective.

Health Checks

Health Checks use endpoint routing with the Generic Host. In `Startup.Configure`, call `MapHealthChecks` on the endpoint builder with the endpoint URL or relative path:

```
C#

app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health");
});
```

Health Checks endpoints can:

- Specify one or more permitted hosts/ports.
- Require authorization.
- Require CORS.

For more information, see [Health checks in ASP.NET Core](#).

Security middleware guidance

Support for authorization and CORS is unified around the [middleware](#) approach. This allows use of the same middleware and functionality across these scenarios. An updated authorization middleware is provided in this release, and CORS Middleware is enhanced so that it can understand the attributes used by MVC controllers.

CORS

Previously, CORS could be difficult to configure. Middleware was provided for use in some use cases, but MVC filters were intended to be used **without** the middleware in other use cases. With ASP.NET Core 3.0, we recommend that all apps that require CORS

use the CORS Middleware in tandem with Endpoint Routing. `UseCors` can be provided with a default policy, and `[EnableCors]` and `[DisableCors]` attributes can be used to override the default policy where required.

In the following example:

- CORS is enabled for all endpoints with the `default` named policy.
- The `MyController` class disables CORS with the `[DisableCors]` attribute.

C#

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

    app.UseRouting();

    app.UseCors("default");

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}

[DisableCors]
public class MyController : ControllerBase
{
    ...
}
```

Authorization

In earlier versions of ASP.NET Core, authorization support was provided via the `[Authorize]` attribute. Authorization middleware wasn't available. In ASP.NET Core 3.0, authorization middleware is required. We recommend placing the ASP.NET Core Authorization Middleware (`UseAuthorization`) immediately after `UseAuthentication`. The Authorization Middleware can also be configured with a default policy, which can be overridden.

In ASP.NET Core 3.0 or later, `UseAuthorization` is called in `Startup.Configure`, and the following `HomeController` requires a signed in user:

C#

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

```

...

app.UseRouting();

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

app.UseEndpoints(endpoints =>
{
    endpoints.MapDefaultControllerRoute();
});
}

public class HomeController : Controller
{
    [Authorize]
    public IActionResult BuyWidgets()
    {
        ...
    }
}

```

When using endpoint routing, we recommend against configuring [AuthorizeFilter](#) and instead relying on the Authorization middleware. If the app uses an `AuthorizeFilter` as a global filter in MVC, we recommend refactoring the code to provide a policy in the call to `AddAuthorization`.

The `DefaultPolicy` is initially configured to require authentication, so no additional configuration is required. In the following example, MVC endpoints are marked as `RequireAuthorization` so that all requests must be authorized based on the `DefaultPolicy`. However, the `HomeController` allows access without the user signing into the app due to `[AllowAnonymous]`:

C#

```

public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

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

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute().RequireAuthorization();
    });
}

```

```
[AllowAnonymous]
public class HomeController : Controller
{
    ...
}
```

Authorization for specific endpoints

Authorization can also be configured for specific classes of endpoints. The following code is an example of converting an MVC app that configured a global `AuthorizeFilter` to an app with a specific policy requiring authorization:

C#

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

    static readonly string _RequireAuthenticatedUserPolicy =
        "RequireAuthenticatedUserPolicy";

    public IConfiguration Configuration { get; }

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

        // Pre 3.0:
        // services.AddMvc(options => options.Filters.Add(new
        // AuthorizeFilter(...)));

        services.AddControllersWithViews();
        services.AddRazorPages();
        services.AddAuthorization(o =>
            o.AddPolicy(_RequireAuthenticatedUserPolicy,
                builder => builder.RequireAuthenticatedUser()));
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
    }
}
```

```

        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

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

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute()
            .RequireAuthorization(_RequireAuthenticatedUserPolicy);
        endpoints.MapRazorPages();
    });
}
}

```

Policies can also be customized. The `DefaultPolicy` is configured to require authentication:

C#

```

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

    public IConfiguration Configuration { get; }

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

        services.AddControllersWithViews();
        services.AddRazorPages();
    }
}

```



```

        services.AddAuthorization(options =>
        {
            options.DefaultPolicy = new AuthorizationPolicyBuilder()
                .RequireAuthenticatedUser()
                .Build();
        });
    }

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

        app.UseRouting();

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

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapDefaultControllerRoute().RequireAuthorization();
            endpoints.MapRazorPages();
        });
    }
}

```

C#

```

[AllowAnonymous]
public class HomeController : Controller
{

```

Alternatively, all endpoints can be configured to require authorization without `[Authorize]` or `RequireAuthorization` by configuring a `FallbackPolicy`. The `FallbackPolicy` is different from the `DefaultPolicy`. The `DefaultPolicy` is triggered by `[Authorize]` or `RequireAuthorization`, while the `FallbackPolicy` is triggered when no other policy is set. `FallbackPolicy` is initially configured to allow requests without authorization.

The following example is the same as the preceding `DefaultPolicy` example but uses the `FallbackPolicy` to always require authentication on all endpoints except when `[AllowAnonymous]` is specified:

C#

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

    services.AddAuthorization(options =>
    {
        options.FallbackPolicy = new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .Build();
    });
}

public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

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

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}

[AllowAnonymous]
public class HomeController : Controller
{
    ...
}
```

Authorization by middleware works without the framework having any specific knowledge of authorization. For instance, [health checks](#) has no specific knowledge of authorization, but health checks can have a configurable authorization policy applied by the middleware.

Additionally, each endpoint can customize its authorization requirements. In the following example, `UseAuthorization` processes authorization with the `DefaultPolicy`, but the `/healthz` health check endpoint requires an `admin` user:

C#

```

public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

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

    app.UseEndpoints(endpoints =>
    {
        endpoints
            .MapHealthChecks("/healthz")
            .RequireAuthorization(new AuthorizeAttribute(){ Roles = "admin",
    });
    });
}

```

Protection is implemented for some scenarios. Endpoints Middleware throws an exception if an authorization or CORS policy is skipped due to missing middleware. Analyzer support to provide additional feedback about misconfiguration is in progress.

Custom authorization handlers

If the app uses custom [authorization handlers](#), endpoint routing passes a different resource type to handlers than MVC. Handlers that expect the authorization handler context resource to be of type [AuthorizationFilterContext](#) (the resource type [provided by MVC filters](#)) will need to be updated to handle resources of type [RouteEndpoint](#) (the resource type given to authorization handlers by endpoint routing).

MVC still uses `AuthorizationFilterContext` resources, so if the app uses MVC authorization filters along with endpoint routing authorization, it may be necessary to handle both types of resources.

SignalR

Mapping of SignalR hubs now takes place inside `UseEndpoints`.

Map each hub with `MapHub`. As in previous versions, each hub is explicitly listed.

In the following example, support for the `ChatHub` SignalR hub is added:

```

C#

public void Configure(IApplicationBuilder app)
{

```

```

...

app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<ChatHub>();
});
}

```

There is a new option for controlling message size limits from clients. For example, in `Startup.ConfigureServices`:

```

C#

services.AddSignalR(hubOptions =>
{
    hubOptions.MaximumReceiveMessageSize = 32768;
});

```

In ASP.NET Core 2.2, you could set the `TransportMaxBufferSize` and that would effectively control the maximum message size. In ASP.NET Core 3.0, that option now only controls the maximum size before backpressure is observed.

SignalR assemblies in shared framework

ASP.NET Core SignalR server-side assemblies are now installed with the .NET Core SDK. For more information, see [Remove obsolete package references](#) in this document.

MVC controllers

Mapping of controllers now takes place inside `UseEndpoints`.

Add `MapControllers` if the app uses attribute routing. Since routing includes support for many frameworks in ASP.NET Core 3.0 or later, adding attribute-routed controllers is opt-in.

Replace the following:

- `MapRoute` with `MapControllerRoute`
- `MapAreaRoute` with `MapAreaControllerRoute`

Since routing now includes support for more than just MVC, the terminology has changed to make these methods clearly state what they do. Conventional routes such as

`MapControllerRoute` / `MapAreaControllerRoute` / `MapDefaultControllerRoute` are applied in the order that they're added. Place more specific routes (such as routes for an area) first.

In the following example:

- `MapControllers` adds support for attribute-routed controllers.
- `MapAreaControllerRoute` adds a conventional route for controllers in an area.
- `MapControllerRoute` adds a conventional route for controllers.

C#

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

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapAreaControllerRoute(
            "admin",
            "admin",
            "Admin/{controller=Home}/{action=Index}/{id?}");
        endpoints.MapControllerRoute(
            "default", "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Async suffix removal from controller action names

In ASP.NET Core 3.0, ASP.NET Core MVC removes the `Async` suffix from controller action names. Both routing and link generation are impacted by this new default. For example:

C#

```
public class ProductsController : Controller
{
    public async Task<IActionResult> ListAsync()
    {
        var model = await _dbContext.Products.ToListAsync();
        return View(model);
    }
}
```

Prior to ASP.NET Core 3.0:

- The preceding action could be accessed at the *Products/ListAsync* route.

- Link generation required specifying the `Async` suffix. For example:

C#HTML

```
<a asp-controller="Products" asp-action="ListAsync">List</a>
```

In ASP.NET Core 3.0:

- The preceding action can be accessed at the *Products/List* route.
- Link generation doesn't require specifying the `Async` suffix. For example:

C#HTML

```
<a asp-controller="Products" asp-action="List">List</a>
```

This change doesn't affect names specified using the `[ActionName]` attribute. The default behavior can be disabled with the following code in `Startup.ConfigureServices`:

C#

```
services.AddMvc(options =>  
    options.SuppressAsyncSuffixInActionNames = false);
```

Changes to link generation

There are some differences in link generation (using `Url.Link` and similar APIs, for example). These include:

- By default, when using endpoint routing, casing of route parameters in generated URIs is not necessarily preserved. This behavior can be controlled with the `IOutboundParameterTransformer` interface.
- Generating a URI for an invalid route (a controller/action or page that doesn't exist) will produce an empty string under endpoint routing instead of producing an invalid URI.
- Ambient values (route parameters from the current context) are not automatically used in link generation with endpoint routing. Previously, when generating a link to another action (or page), unspecified route values would be inferred from the *current* routes ambient values. When using endpoint routing, all route parameters must be specified explicitly during link generation.

Razor Pages

Mapping Razor Pages now takes place inside `UseEndpoints`.

Add `MapRazorPages` if the app uses Razor Pages. Since Endpoint Routing includes support for many frameworks, adding Razor Pages is now opt-in.

In the following `Startup.Configure` method, `MapRazorPages` adds support for Razor Pages:

C#

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

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Use MVC without Endpoint Routing

Using MVC via `UseMvc` or `UseMvcWithDefaultRoute` in ASP.NET Core 3.0 requires an explicit opt-in inside `Startup.ConfigureServices`. This is required because MVC must know whether it can rely on the authorization and CORS Middleware during initialization. An analyzer is provided that warns if the app attempts to use an unsupported configuration.

If the app requires legacy `IRouter` support, disable `EnableEndpointRouting` using any of the following approaches in `Startup.ConfigureServices`:

C#

```
services.AddMvc(options => options.EnableEndpointRouting = false);
```

C#

```
services.AddControllers(options => options.EnableEndpointRouting = false);
```

C#

```
services.AddControllersWithViews(options => options.EnableEndpointRouting =
```

```
false);
```

C#

```
services.AddRazorPages().AddMvcOptions(options =>  
options.EnableEndpointRouting = false);
```

Health checks

Health checks can be used as a *router-ware* with Endpoint Routing.

Add `MapHealthChecks` to use health checks with Endpoint Routing. The `MapHealthChecks` method accepts arguments similar to `UseHealthChecks`. The advantage of using `MapHealthChecks` over `UseHealthChecks` is the ability to apply authorization and to have greater fine-grained control over the matching policy.

In the following example, `MapHealthChecks` is called for a health check endpoint at `/healthz`:

C#

```
public void Configure(IApplicationBuilder app)  
{  
    ...  
    app.UseRouting();  
    app.UseEndpoints(endpoints =>  
    {  
        endpoints.MapHealthChecks("/healthz", new HealthCheckOptions() { });  
    });  
}
```

HostBuilder replaces WebHostBuilder

The ASP.NET Core 3.0 templates use [Generic Host](#). Previous versions used [Web Host](#). The following code shows the ASP.NET Core 3.0 template generated `Program` class:

C#

```
// requires using Microsoft.AspNetCore.Hosting;  
// requires using Microsoft.Extensions.Hosting;  
  
public class Program  
{
```



```

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

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

```

The following code shows the ASP.NET Core 2.2 template-generated `Program` class:

```

C#

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

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

```

`IWebHostBuilder` remains in 3.0 and is the type of the `webBuilder` seen in the preceding code sample. `WebHostBuilder` will be deprecated in a future release and replaced by `HostBuilder`.

The most significant change from `WebHostBuilder` to `HostBuilder` is in [dependency injection \(DI\)](#). When using `HostBuilder`, you can only inject the following into `Startup`'s constructor:

- [IConfiguration](#)
- [IHostEnvironment](#)
- [IWebHostEnvironment](#)

The `HostBuilder` DI constraints:

- Enable the DI container to be built only one time.
- Avoids the resulting object lifetime issues like resolving multiple instances of singletons.

For more information, see [Avoiding Startup service injection in ASP.NET Core 3](#).

AddAuthorization moved to a different assembly

The ASP.NET Core 2.2 and lower `AddAuthorization` methods in *Microsoft.AspNetCore.Authorization.dll*:

- Have been renamed `AddAuthorizationCore`.
- Have been moved to *Microsoft.AspNetCore.Authorization.Policy.dll*.

Apps that are using both *Microsoft.AspNetCore.Authorization.dll* and *Microsoft.AspNetCore.Authorization.Policy.dll* aren't impacted.

Apps that are not using *Microsoft.AspNetCore.Authorization.Policy.dll* should do one of the following:

- Add a reference to *Microsoft.AspNetCore.Authorization.Policy.dll*. This approach works for most apps and is all that is required.
- Switch to using `AddAuthorizationCore`

For more information, see [Breaking change in AddAuthorization\(o =>\) overload lives in a different assembly #386](#).

Identity UI

Identity UI updates for ASP.NET Core 3.0:

- Add a package reference to [Microsoft.AspNetCore.Identity.UI](#).
- Apps that don't use Razor Pages must call `MapRazorPages`. See [Razor Pages](#) in this document.
- Bootstrap 4 is the default UI framework. Set an `IdentityUIFrameworkVersion` project property to change the default. For more information, see [this GitHub announcement](#).

SignalR

The SignalR JavaScript client has changed from `@aspnet/signalr` to `@microsoft/signalr`. To react to this change, change the references in `package.json` files, `require` statements, and ECMAScript `import` statements.

System.Text.Json is the default protocol

`System.Text.Json` is now the default Hub protocol used by both the client and server.

In `Startup.ConfigureServices`, call `AddJsonProtocol` to set serializer options.

Server:

```
C#

services.AddSignalR(...)
    .AddJsonProtocol(options =>
    {
        options.PayloadSerializerOptions.WriteIndented = false;
    })
```

Client:

```
C#

new HubConnectionBuilder()
    .WithUrl("/chathub")
    .AddJsonProtocol(options =>
    {
        options.PayloadSerializerOptions.WriteIndented = false;
    })
    .Build();
```

Switch to Newtonsoft.Json

If you're using [features of Newtonsoft.Json](#) that aren't supported in `System.Text.Json`, you can switch back to `Newtonsoft.Json`. See [Use Newtonsoft.Json in an ASP.NET Core 3.0 SignalR project](#) earlier in this article.

Redis distributed caches

The [Microsoft.Extensions.Caching.Redis](#) package isn't available for ASP.NET Core 3.0 or later apps. Replace the package reference with [Microsoft.Extensions.Caching.StackExchangeRedis](#). For more information, see [Distributed caching in ASP.NET Core](#).


Opt in to runtime compilation

Prior to ASP.NET Core 3.0, runtime compilation of views was an implicit feature of the framework. Runtime compilation supplements build-time compilation of views. It allows

the framework to compile Razor views and pages (`.cshtml` files) when the files are modified, without having to rebuild the entire app. This feature supports the scenario of making a quick edit in the IDE and refreshing the browser to view the changes.

In ASP.NET Core 3.0, runtime compilation is an opt-in scenario. Build-time compilation is the only mechanism for view compilation that's enabled by default. The runtime relies on Visual Studio or [dotnet-watch](#) in Visual Studio Code to rebuild the project when it detects changes to `.cshtml` files. In Visual Studio, changes to `.cs`, `.cshtml`, or `.razor` files in the project being run (`Ctrl+F5`), but not debugged (`F5`), trigger recompilation of the project.

To enable runtime compilation in your ASP.NET Core 3.0 project:

1. Install the [Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation](#)  NuGet package.
2. Update `Startup.ConfigureServices` to call `AddRazorRuntimeCompilation`:

For ASP.NET Core MVC, use the following code:

```
C#


services.AddControllersWithViews()
    .AddRazorRuntimeCompilation(...);
```

For ASP.NET Core Razor Pages, use the following code:

```
C#

services.AddRazorPages()
    .AddRazorRuntimeCompilation(...);
```

The sample at

<https://github.com/aspnet/samples/tree/main/samples/aspnetcore/mvc/runtimecompilation>  shows an example of enabling runtime compilation conditionally in Development environments.

For more information on Razor file compilation, see [Razor file compilation in ASP.NET Core](#).

Migrate libraries via multi-targeting

Libraries often need to support multiple versions of ASP.NET Core. Most libraries that were compiled against previous versions of ASP.NET Core should continue working without issues. The following conditions require the app to be cross-compiled:

- The library relies on a feature that has a binary [breaking change](#).
- The library wants to take advantage of new features in ASP.NET Core 3.0.

For example:

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFrameworks>netcoreapp3.0;netstandard2.0</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.0'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netstandard2.0'">
    <PackageReference Include="Microsoft.AspNetCore" Version="2.1.0" />
  </ItemGroup>
</Project>
```

Use `#ifdefs` to enable ASP.NET Core 3.0-specific APIs:

C#

```
var webRootFileProvider =
#if NETCOREAPP3_0
    GetRequiredService<IWebHostEnvironment>().WebRootFileProvider;
#elif NETSTANDARD2_0
    GetRequiredService<IHostingEnvironment>().WebRootFileProvider;
#else
#error unknown target framework
#endif
```

For more information on using ASP.NET Core APIs in a class library, see [Use ASP.NET Core APIs in a class library](#).

Miscellaneous changes

The validation system in .NET Core 3.0 and later treats non-nullable parameters or bound properties as if they had a `[Required]` attribute. For more information, see [\[Required\] attribute](#).

Publish

Delete the *bin* and *obj* folders in the project directory.

TestServer

For apps that use [TestServer](#) directly with the [Generic Host](#), create the `TestServer` on an [IWebHostBuilder](#) in [ConfigureWebHost](#):

C#

```
[Fact]
public async Task GenericCreateAndStartHost_GetTestServer()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .Configure(app => { });
        })
        .StartAsync();

    var response = await host.GetTestServer().CreateClient().GetAsync("/");

    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}
```

Breaking API changes

Review breaking changes:

- [Complete list of breaking changes in the ASP.NET Core 3.0 release](#) ↗
- [Breaking API changes in Antiforgery, CORS, Diagnostics, MVC, and Routing](#) ↗. This list includes breaking changes for compatibility switches.
- For a summary of 2.2-to-3.0 breaking changes across .NET Core, ASP.NET Core, and Entity Framework Core, see [Breaking changes for migration from version 2.2 to 3.0](#).

Endpoint routing with catch-all parameter

.NET Core 3.0 on Azure App Service

The rollout of .NET Core to Azure App Service is finished. .NET Core 3.0 is available in all Azure App Service datacenters.

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](#).

Migrate from ASP.NET Core 2.1 to 2.2

Article • 06/18/2024

By [Scott Addie](#) 


This article explains how to update an existing ASP.NET Core 2.1 project to ASP.NET Core 2.2.

Prerequisites

Visual Studio

- [Visual Studio 2019](#) with the **ASP.NET and web development** workload
- [.NET Core SDK 2.2 or later](#) 

Warning

If you use Visual Studio 2017, see [dotnet/sdk issue #3124](#)  for information about .NET Core SDK versions that don't work with Visual Studio.

Update Target Framework Moniker (TFM)

Projects targeting .NET Core should use the [TFM](#) of a version greater than or equal to .NET Core 2.2. In the project file, update the `<TargetFramework>` node's inner text with `netcoreapp2.2`:

XML

```
<TargetFramework>netcoreapp2.2</TargetFramework>
```

Projects targeting .NET Framework may continue to use the TFM of a version greater than or equal to .NET Framework 4.6.1:

XML

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