

What's new in ASP.NET Core 6.0

Article • 11/02/2023

This article highlights the most significant changes in ASP.NET Core 6.0 with links to relevant documentation.

ASP.NET Core MVC and Razor improvements

Minimal APIs

Minimal APIs are architected to create HTTP APIs with minimal dependencies. They are ideal for microservices and apps that want to include only the minimum files, features, and dependencies in ASP.NET Core. For more information, see:

- [Tutorial: Create a minimal API with ASP.NET Core](#)
- [Differences between minimal APIs and APIs with controllers](#)
- [Minimal APIs quick reference](#)
- [Code samples migrated to the new minimal hosting model in 6.0](#)

SignalR

Long running activity tag for SignalR connections

SignalR uses the new [Microsoft.AspNetCore.Http.Features.IHttpActivityFeature.Activity](#) to add an `http.long_running` tag to the request activity.

`IHttpActivityFeature.Activity` is used by [APM services](#) like [Azure Monitor](#) [Application Insights](#) to filter SignalR requests from creating long running request alerts.

SignalR performance improvements

- Allocate [HubCallerClients](#) once per connection instead of every hub method call.
- Avoid closure allocation in SignalR `DefaultHubDispatcher.Invoke`. State is passed to a local static function via parameters to avoid a closure allocation. For more information, see [this GitHub pull request](#).
- Allocate a single [StreamItemMessage](#) per stream instead of per stream item in server-to-client streaming. For more information, see [this GitHub pull request](#).

Razor compiler

Razor compiler updated to use source generators

The Razor compiler is now based on [C# source generators](#). Source generators run during compilation and inspect what is being compiled to produce additional files that are compiled along with the rest of the project. Using source generators simplifies the Razor compiler and significantly speeds up build times.

Razor compiler no longer produces a separate Views assembly

The Razor compiler previously utilized a two-step compilation process that produced a separate Views assembly that contained the generated views and pages (`.cshtml` files) defined in the app. The generated types were public and under the `AspNetCore` namespace.

The updated Razor compiler builds the views and pages types into the main project assembly. These types are now generated by default as internal sealed in the `AspNetCoreGeneratedDocument` namespace. This change improves build performance, enables single file deployment, and enables these types to participate in [Hot Reload](#).

For more information about this change, see [the related announcement issue](#) [↗](#) on GitHub.

ASP.NET Core performance and API improvements

Many changes were made to reduce allocations and improve performance across the stack:

- Non-allocating [app.Use](#) extension method. The new overload of `app.Use` requires passing the context to `next` which saves two internal per-request allocations that are required when using the other overload.
- Reduced memory allocations when accessing [HttpRequest.Cookies](#). For more information, see [this GitHub issue](#) [↗](#).
- Use [LoggerMessage.Define](#) for the windows only [HTTP.sys web server](#). The `ILogger` extension methods calls have been replaced with calls to `LoggerMessage.Define`.

- Reduce the per connection overhead in [SocketConnection](#) by ~30%. For more information, see [this GitHub pull request](#).
- Reduce allocations by removing logging delegates in generic types. For more information, see [this GitHub pull request](#).
- Faster GET access (about 50%) to commonly-used features such as [IHttpRequestFeature](#), [IHttpResponseFeature](#), [IHttpResponseBodyFeature](#), [IRouteValuesFeature](#), and [IEndpointFeature](#). For more information, see [this GitHub pull request](#).
- Use single instance strings for known header names, even if they aren't in the preserved header block. Using single instance string helps prevent multiple duplicates of the same string in long lived connections, for example, in [Microsoft.AspNetCore.WebSockets](#). For more information, see [this GitHub issue](#).
- Reuse [HttpProtocol CancellationTokenSource](#) in Kestrel. Use the new [CancellationTokenSource.TryReset](#) method on `CancellationTokenSource` to reuse tokens if they haven't been canceled. For more information, see [this GitHub issue](#) and this [video](#).
- Implement and use an [AdaptiveCapacityDictionary](#) in [Microsoft.AspNetCore.Http.RequestCookieCollection](#) for more efficient access to dictionaries. For more information, see [this GitHub pull request](#).

Reduced memory footprint for idle TLS connections

For long running TLS connections where data is only occasionally sent back and forth, we've significantly reduced the memory footprint of ASP.NET Core apps in .NET 6. This should help improve the scalability of scenarios such as WebSocket servers. This was possible due to numerous improvements in [System.IO.Pipelines](#), [SslStream](#), and Kestrel. The following sections detail some of the improvements that have contributed to the reduced memory footprint:

Reduce the size of `System.IO.Pipelines.Pipe`

For every connection that is established, two pipes are allocated in Kestrel:

- The transport layer to the app for the request.
- The application layer to the transport for the response.

By shrinking the size of [System.IO.Pipelines.Pipe](#) from 368 bytes to 264 bytes (about a 28.2% reduction), 208 bytes per connection are saved (104 bytes per Pipe).

Pool SocketSender

`SocketSender` objects (that subclass `SocketAsyncEventArgs`) are around 350 bytes at runtime. Instead of allocating a new `SocketSender` object per connection, they can be pooled. `SocketSender` objects can be pooled because sends are usually very fast. Pooling reduces the per connection overhead. Instead of allocating 350 bytes per connection, only pay 350 bytes per `IOQueue` are allocated. Allocation is done per queue to avoid contention. Our WebSocket server with 5000 idle connections went from allocating ~1.75 MB (350 bytes * 5000) to allocating ~2.8 kb (350 bytes * 8) for `SocketSender` objects.

Zero bytes reads with `SslStream`

Bufferless reads are a technique employed in ASP.NET Core to avoid renting memory from the memory pool if there's no data available on the socket. Prior to this change, our WebSocket server with 5000 idle connections required ~200 MB without TLS compared to ~800 MB with TLS. Some of these allocations (4k per connection) were from Kestrel having to hold on to an `ArrayPool<T>` buffer while waiting for the reads on `SslStream` to complete. Given that these connections were idle, none of reads completed and returned their buffers to the `ArrayPool`, forcing the `ArrayPool` to allocate more memory. The remaining allocations were in `SslStream` itself: 4k buffer for TLS handshakes and 32k buffer for normal reads. In .NET 6, when the user performs a zero byte read on `SslStream` and it has no data available, `SslStream` internally performs a zero-byte read on the underlying wrapped stream. In the best case (idle connection), these changes result in a savings of 40 Kb per connection while still allowing the consumer (Kestrel) to be notified when data is available without holding on to any unused buffers.

Zero byte reads with `PipeReader`

With bufferless reads supported on `SslStream`, an option was added to perform zero byte reads to `StreamPipeReader`, the internal type that adapts a `Stream` into a `PipeReader`. In Kestrel, a `StreamPipeReader` is used to adapt the underlying `SslStream` into a `PipeReader`. Therefore it was necessary to expose these zero byte read semantics on the `PipeReader`.

A `PipeReader` can now be created that supports zero bytes reads over any underlying `Stream` that supports zero byte read semantics (e.g., `SslStream`, `NetworkStream`, etc) using the following API:

```
.NET CLI
```

```
var reader = PipeReader.Create(stream, new  
StreamPipeReaderOptions(useZeroByteReads: true));
```

Remove slabs from the `SlabMemoryPool`

To reduce fragmentation of the heap, Kestrel employed a technique where it allocated slabs of memory of 128 KB as part of its memory pool. The slabs were then further divided into 4 KB blocks that were used by Kestrel internally. The slabs had to be larger than 85 KB to force allocation on the large object heap to try and prevent the GC from relocating this array. However, with the introduction of the new GC generation, [Pinned Object Heap](#) (POH), it no longer makes sense to allocate blocks on slab. Kestrel now directly allocates blocks on the POH, reducing the complexity involved in managing the memory pool. This change should make easier to perform future improvements such as making it easier to shrink the memory pool used by Kestrel.

`IAsyncDisposable` supported

[IAsyncDisposable](#) is now available for controllers, Razor Pages, and View Components. Asynchronous versions have been added to the relevant interfaces in factories and activators:

- The new methods offer a default interface implementation that delegates to the synchronous version and calls [Dispose](#).
- The implementations override the default implementation and handle disposing `IAsyncDisposable` implementations.
- The implementations favor `IAsyncDisposable` over `IDisposable` when both interfaces are implemented.
- Extenders must override the new methods included to support `IAsyncDisposable` instances.

`IAsyncDisposable` is beneficial when working with:

- Asynchronous enumerators, for example, in asynchronous streams.
- Unmanaged resources that have resource-intensive I/O operations to release.

When implementing this interface, use the `DisposeAsync` method to release resources.

Consider a controller that creates and uses a [Utf8JsonWriter](#). `Utf8JsonWriter` is an `IAsyncDisposable` resource:

```

public class HomeController : Controller, IAsyncDisposable
{
    private Utf8JsonWriter? _jsonWriter;
    private readonly ILogger<HomeController> _logger;

    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
        _jsonWriter = new Utf8JsonWriter(new MemoryStream());
    }
}

```

`IAsyncDisposable` must implement `DisposeAsync`:

```

C#

public async ValueTask DisposeAsync()
{
    if (_jsonWriter is not null)
    {
        await _jsonWriter.DisposeAsync();
    }

    _jsonWriter = null;
}

```

Vcpkg port for SignalR C++ client

[Vcpkg](#) is a cross-platform command-line package manager for C and C++ libraries. We've recently added a port to `vcpkg` to add `CMake` native support for the SignalR C++ client. `vcpkg` also works with MSBuild.

The SignalR client can be added to a CMake project with the following snippet when the `vcpkg` is included in the toolchain file:

```

.NET CLI

find_package(microsoft-signalr CONFIG REQUIRED)
link_libraries(microsoft-signalr::microsoft-signalr)



```

With the preceding snippet, the SignalR C++ client is ready to use `#include` and used in a project without any additional configuration. For a complete example of a C++ application that utilizes the SignalR C++ client, see the [halter73/SignalR-Client-Cpp-Sample](#) repository.

Blazor

Project template changes

Several project template changes were made for Blazor apps, including the use of the `Pages/_Layout.cshtml` file for layout content that appeared in the `_Host.cshtml` file for earlier Blazor Server apps. Study the changes by creating an app from a 6.0 project template or accessing the ASP.NET Core reference source for the project templates:

- [Blazor Server](#) 
- [Blazor WebAssembly](#) 

Blazor WebAssembly native dependencies support

Blazor WebAssembly apps can use native dependencies built to run on WebAssembly. For more information, see [ASP.NET Core Blazor WebAssembly native dependencies](#).

WebAssembly Ahead-of-time (AOT) compilation and runtime relinking

Blazor WebAssembly supports ahead-of-time (AOT) compilation, where you can compile your .NET code directly into WebAssembly. AOT compilation results in runtime performance improvements at the expense of a larger app size. Relinking the .NET WebAssembly runtime trims unused runtime code and thus improves download speed. For more information, see [Ahead-of-time \(AOT\) compilation](#) and [Runtime relinking](#).

Persist prerendered state

Blazor supports persisting state in a prerendered page so that the state doesn't need to be recreated when the app is fully loaded. For more information, see [Prerender and integrate ASP.NET Core Razor components](#).

Error boundaries

Error boundaries provide a convenient approach for handling exceptions on the UI level. For more information, see [Handle errors in ASP.NET Core Blazor apps](#).

SVG support

The `<foreignObject>` [element](#) is supported to display arbitrary HTML within an SVG. For more information, see [ASP.NET Core Razor components](#).

Blazor Server support for byte array transfer in JS Interop

Blazor supports optimized byte array JS interop that avoids encoding and decoding byte arrays into Base64. For more information, see the following resources:

- [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#)
- [Call .NET methods from JavaScript functions in ASP.NET Core Blazor](#)

Query string enhancements

Support for working with query strings is improved. For more information, see [ASP.NET Core Blazor routing and navigation](#).

Binding to select multiple

Binding supports multiple option selection with `<input>` elements. For more information, see the following resources:

- [ASP.NET Core Blazor data binding](#)
- [ASP.NET Core Blazor input components](#)

Head (`<head>`) content control

Razor components can modify the HTML `<head>` element content of a page, including setting the page's title (`<title>` element) and modifying metadata (`<meta>` elements). For more information, see [Control `<head>` content in ASP.NET Core Blazor apps](#).

Generate Angular and React components

Generate framework-specific JavaScript components from Razor components for web frameworks, such as Angular or React. For more information, see [ASP.NET Core Razor components](#).

Render components from JavaScript

Render Razor components dynamically from JavaScript for existing JavaScript apps. For more information, see [ASP.NET Core Razor components](#).

Custom elements

Experimental support is available for building custom elements, which use standard HTML interfaces. For more information, see [ASP.NET Core Razor components](#).

Infer component generic types from ancestor components

An ancestor component can cascade a type parameter by name to descendants using the new `[CascadingTypeParameter]` attribute. For more information, see [ASP.NET Core Razor components](#).

Dynamically rendered components

Use the new built-in `DynamicComponent` component to render components by type. For more information, see [Dynamically-rendered ASP.NET Core Razor components](#).

Improved Blazor accessibility

Use the new `FocusOnNavigate` component to set the UI focus to an element based on a CSS selector after navigating from one page to another. For more information, see [ASP.NET Core Blazor routing and navigation](#).

Custom event argument support

Blazor supports custom event arguments, which enable you to pass arbitrary data to .NET event handlers with custom events. For more information, see [ASP.NET Core Blazor event handling](#).

Required parameters

Apply the new `[EditorRequired]` attribute to specify a required component parameter. For more information, see [ASP.NET Core Razor components](#).

Collocation of JavaScript files with pages, views, and components

Collocate JavaScript files for pages, views, and Razor components as a convenient way to organize scripts in an app. For more information, see [ASP.NET Core Blazor JavaScript](#)

[interoperability \(JS interop\)](#).

JavaScript initializers

JavaScript initializers execute logic before and after a Blazor app loads. For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

Streaming JavaScript interop

Blazor now supports streaming data directly between .NET and JavaScript. For more information, see the following resources:

- [Stream from .NET to JavaScript](#)
- [Stream from JavaScript to .NET](#)

Generic type constraints

Generic type parameters are now supported. For more information, see [ASP.NET Core Razor components](#).

WebAssembly deployment layout

Use a deployment layout to enable Blazor WebAssembly app downloads in restricted security environments. For more information, see [Deployment layout for ASP.NET Core hosted Blazor WebAssembly apps](#).

New Blazor articles

In addition to the Blazor features described in the preceding sections, new Blazor articles are available on the following subjects:

- [ASP.NET Core Blazor file downloads](#): Learn how to download a file using native `byte[]` streaming interop to ensure efficient transfer to the client.
- [Work with images in ASP.NET Core Blazor](#): Discover how to work with images in Blazor apps, including how to stream image data and preview an image.

Build Blazor Hybrid apps with .NET MAUI, WPF, and Windows Forms


Use Blazor Hybrid to blend desktop and mobile native client frameworks with .NET and Blazor:

- .NET Multi-platform App UI (.NET MAUI) is a cross-platform framework for creating native mobile and desktop apps with C# and XAML.
- Blazor Hybrid apps can be built with Windows Presentation Foundation (WPF) and Windows Forms frameworks.


Important


Blazor Hybrid is in preview and shouldn't be used in production apps until final release.

For more information, see the following resources:

- [Preview ASP.NET Core Blazor Hybrid documentation](#)
- [What is .NET MAUI?](#)
- [Microsoft .NET Blog \(category: ".NET MAUI"\)](#)

Kestrel

[HTTP/3](#) is currently in draft and therefore subject to change. HTTP/3 support in ASP.NET Core is not released, it's a preview feature included in .NET 6.

Kestrel now supports HTTP/3. For more information, see [Use HTTP/3 with the ASP.NET Core Kestrel web server](#) and the blog entry [HTTP/3 support in .NET 6](#).

New Kestrel logging categories for selected logging

Prior to this change, enabling verbose logging for Kestrel was prohibitively expensive as all of Kestrel shared the `Microsoft.AspNetCore.Server.Kestrel` logging category name. `Microsoft.AspNetCore.Server.Kestrel` is still available, but the following new subcategories allow for more control of logging:

- `Microsoft.AspNetCore.Server.Kestrel` (current category): `ApplicationError`, `ConnectionHeadResponseBodyWrite`, `ApplicationNeverCompleted`, `RequestBodyStart`, `RequestBodyDone`, `RequestBodyNotEntirelyRead`, `RequestBodyDrainTimedOut`, `ResponseMinimumDataRateNotSatisfied`, `InvalidResponseHeaderRemoved`, `HeartbeatSlow`.
- `Microsoft.AspNetCore.Server.Kestrel.BadRequests`: `ConnectionBadRequest`, `RequestProcessingError`, `RequestBodyMinimumDataRateNotSatisfied`.

- `Microsoft.AspNetCore.Server.Kestrel.Connections`: `ConnectionAccepted`, `ConnectionStart`, `ConnectionStop`, `ConnectionPause`, `ConnectionResume`, `ConnectionKeepAlive`, `ConnectionRejected`, `ConnectionDisconnect`, `NotAllConnectionsClosedGracefully`, `NotAllConnectionsAborted`, `ApplicationAbortedConnection`.
- `Microsoft.AspNetCore.Server.Kestrel.Http2`: `Http2ConnectionError`, `Http2ConnectionClosing`, `Http2ConnectionClosed`, `Http2StreamError`, `Http2StreamResetAbort`, `HPackDecodingError`, `HPackEncodingError`, `Http2FrameReceived`, `Http2FrameSending`, `Http2MaxConcurrentStreamsReached`.
- `Microsoft.AspNetCore.Server.Kestrel.Http3`: `Http3ConnectionError`, `Http3ConnectionClosing`, `Http3ConnectionClosed`, `Http3StreamAbort`, `Http3FrameReceived`, `Http3FrameSending`.

Existing rules continue to work, but you can now be more selective on which rules you enable. For example, the observability overhead of enabling `Debug` logging for just bad requests is greatly reduced and can be enabled with the following configuration:

XML

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.Kestrel.BadRequests": "Debug"
    }
  }
}
```

Log filtering applies rules with the longest matching category prefix. For more information, see [How filtering rules are applied](#)

Emit KestrelServerOptions via EventSource event

The [KestrelEventSource](#) emits a new event containing the JSON-serialized [KestrelServerOptions](#) when enabled with verbosity `EventLevel.LogAlways`. This event makes it easier to reason about the server behavior when analyzing collected traces. The following JSON is an example of the event payload:

JSON

```
{
  "AllowSynchronousIO": false,
  "AddServerHeader": true,
```

```
"AllowAlternateSchemes": false,
"AllowResponseHeaderCompression": true,
"EnableAltSvc": false,
"IsDevCertLoaded": true,
"RequestHeaderEncodingSelector": "default",
"ResponseHeaderEncodingSelector": "default",
"Limits": {
  "KeepAliveTimeout": "00:02:10",
  "MaxConcurrentConnections": null,
  "MaxConcurrentUpgradedConnections": null,
  "MaxRequestBodySize": 30000000,
  "MaxRequestBufferSize": 1048576,
  "MaxRequestHeaderCount": 100,
  "MaxRequestHeadersTotalSize": 32768,
  "MaxRequestLineSize": 8192,
  "MaxResponseBufferSize": 65536,
  "MinRequestBodyDataRate": "Bytes per second: 240, Grace Period:
00:00:05",
  "MinResponseDataRate": "Bytes per second: 240, Grace Period:
00:00:05",
  "RequestHeadersTimeout": "00:00:30",
  "Http2": {
    "MaxStreamsPerConnection": 100,
    "HeaderTableSize": 4096,
    "MaxFrameSize": 16384,
    "MaxRequestHeaderFieldSize": 16384,
    "InitialConnectionWindowSize": 131072,
    "InitialStreamWindowSize": 98304,
    "KeepAlivePingDelay": "10675199.02:48:05.4775807",
    "KeepAlivePingTimeout": "00:00:20"
  },
  "Http3": {
    "HeaderTableSize": 0,
    "MaxRequestHeaderFieldSize": 16384
  }
},
"ListenOptions": [
  {
    "Address": "https://127.0.0.1:7030",
    "IsTls": true,
    "Protocols": "Http1AndHttp2"
  },
  {
    "Address": "https://[::1]:7030",
    "IsTls": true,
    "Protocols": "Http1AndHttp2"
  },
  {
    "Address": "http://127.0.0.1:5030",
    "IsTls": false,
    "Protocols": "Http1AndHttp2"
  },
  {
    "Address": "http://[::1]:5030",
    "IsTls": false,
```

```
        "Protocols": "Http1AndHttp2"
    }
}
}
```

New DiagnosticSource event for rejected HTTP requests

Kestrel now emits a new `DiagnosticSource` event for HTTP requests rejected at the server layer. Prior to this change, there was no way to observe these rejected requests. The new `DiagnosticSource` event

`Microsoft.AspNetCore.Server.Kestrel.BadRequest` contains a `IBadRequestExceptionFeature` that can be used to introspect the reason for rejecting the request.

C#

```
using Microsoft.AspNetCore.Http.Features;
using System.Diagnostics;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
var diagnosticSource = app.Services.GetRequiredService<DiagnosticListener>();
using var badRequestListener = new
BadRequestEventListener(diagnosticSource,
    (badRequestExceptionFeature) =>
    {
        app.Logger.LogError(badRequestExceptionFeature.Error, "Bad request received");
    });
app.MapGet("/", () => "Hello world");

app.Run();

class BadRequestEventListener : IObservable<KeyValuePair<string, object>>, IDisposable
{
    private readonly IDisposable _subscription;
    private readonly Action<IBadRequestExceptionFeature> _callback;

    public BadRequestEventListener(DiagnosticListener diagnosticListener,
        Action<IBadRequestExceptionFeature> callback)
    {
        _subscription = diagnosticListener.Subscribe(this!,
            IsEnabled);
        _callback = callback;
    }

    private static readonly Predicate<string> IsEnabled = (provider)
```

```

=> provider switch
{
    "Microsoft.AspNetCore.Server.Kestrel.BadRequest" => true,
    _ => false
};
public void OnNext(KeyValuePair<string, object> pair)
{
    if (pair.Value is IFeatureCollection featureCollection)
    {
        var badRequestFeature = featureCollection.Get<IBadRequestExceptionFeature>();

        if (badRequestFeature is not null)
        {
            _callback(badRequestFeature);
        }
    }
}
public void OnError(Exception error) { }
public void OnCompleted() { }
public virtual void Dispose() => _subscription.Dispose();
}

```

For more information, see [Logging and diagnostics in Kestrel](#).

Create a ConnectionContext from an Accept Socket

The new [SocketConnectionContextFactory](#) makes it possible to create a [ConnectionContext](#) from an accepted socket. This makes it possible to build a custom socket-based [IConnectionListenerFactory](#) without losing out on all the performance work and pooling happening in [SocketConnection](#) [↗].

See [this example of a custom IConnectionListenerFactory](#) [↗] which shows how to use this `SocketConnectionContextFactory`.

Kestrel is the default launch profile for Visual Studio

The default launch profile for all new dotnet web projects is Kestrel. Starting Kestrel is significantly faster and results in a more responsive experience while developing apps.

IIS Express is still available as a launch profile for scenarios such as Windows Authentication or port sharing.

Localhost ports for Kestrel are random

See [Template generated ports for Kestrel](#) in this document for more information.

Authentication and authorization

Authentication servers

.NET 3 to .NET 5 used [IdentityServer4](#) as part of our template to support the issuing of JWT tokens for SPA and Blazor applications. The templates now use the [Duende Identity Server](#).

If you are extending the identity models and are updating existing projects you need to update the namespaces in your code from `IdentityServer4.IdentityServer` to `Duende.IdentityServer` and follow their [migration instructions](#).

The license model for Duende Identity Server has changed to a reciprocal license, which may require license fees when it's used commercially in production. See the [Duende license page](#) for more details.

Delayed client certificate negotiation

Developers can now opt-in to using delayed client certificate negotiation by specifying [ClientCertificateMode.DelayCertificate](#) on the [HttpsConnectionAdapterOptions](#). This only works with HTTP/1.1 connections because HTTP/2 forbids delayed certificate renegotiation. The caller of this API must buffer the request body before requesting the client certificate:

C#

```
using Microsoft.AspNetCore.Server.Kestrel.Https;
using Microsoft.AspNetCore.WebUtilities;

var builder = WebApplication.CreateBuilder(args);
builder.WebHost.UseKestrel(options =>
{
    options.ConfigureHttpsDefaults(adapterOptions =>
    {
        adapterOptions.ClientCertificateMode =
ClientCertificateMode.DelayCertificate;
    });
});

var app = builder.Build();
app.Use(async (context, next) =>
{
    bool desiredState = GetDesiredState();
```



```
// Check if your desired criteria is met
if (desiredState)
{
    // Buffer the request body
    context.Request.EnableBuffering();
    var body = context.Request.Body;
    await body.DrainAsync(context.RequestAborted);
    body.Position = 0;

    // Request client certificate
    var cert = await
context.Connection.GetClientCertificateAsync();

    // Disable buffering on future requests if the client
    // doesn't provide a cert
}
await next(context);
});

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

OnCheckSlidingExpiration event for controlling cookie renewal

Cookie authentication sliding expiration can now be customized or suppressed using the new [OnCheckSlidingExpiration](#). For example, this event can be used by a single-page app that needs to periodically ping the server without affecting the authentication session.

Miscellaneous

Hot Reload

Quickly make UI and code updates to running apps without losing app state for faster and more productive developer experience using [Hot Reload](#). For more information, see [.NET Hot Reload support for ASP.NET Core](#) and [Update on .NET Hot Reload progress and Visual Studio 2022 Highlights](#) [↗](#).

Improved single-page app (SPA) templates

The ASP.NET Core project templates have been updated for Angular and React to use an improved pattern for single-page apps that is more flexible and more closely aligns

with common patterns for modern front-end web development.

Previously, the ASP.NET Core template for Angular and React used specialized middleware during development to launch the development server for the front-end framework and then proxy requests from ASP.NET Core to the development server. The logic for launching the front-end development server was specific to the command-line interface for the corresponding front-end framework. Supporting additional front-end frameworks using this pattern meant adding additional logic to ASP.NET Core.

The updated ASP.NET Core templates for Angular and React in .NET 6 flips this arrangement around and take advantage of the built-in proxying support in the development servers of most modern front-end frameworks. When the ASP.NET Core app is launched, the front-end development server is launched just as before, but the development server is configured to proxy requests to the backend ASP.NET Core process. All of the front-end specific configuration to setup proxying is part of the app, not ASP.NET Core. Setting up ASP.NET Core projects to work with other front-end frameworks is now straight-forward: setup the front-end development server for the chosen framework to proxy to the ASP.NET Core backend using the pattern established in the Angular and React templates.

The startup code for the ASP.NET Core app no longer needs any single-page app-specific logic. The logic for starting the front-end development server during development is injecting into the app at runtime by the new [Microsoft.AspNetCore.SpaProxy](#) package. Fallback routing is handled using endpoint routing instead of SPA-specific middleware.

Templates that follow this pattern can still be run as a single project in Visual Studio or using `dotnet run` from the command-line. When the app is published, the front-end code in the *ClientApp* folder is built and collected as before into the web root of the host ASP.NET Core app and served as static files. Scripts included in the template configure the front-end development server to use HTTPS using the ASP.NET Core development certificate.

Draft HTTP/3 support in .NET 6

[HTTP/3](#) is currently in draft and therefore subject to change. HTTP/3 support in ASP.NET Core is not released, it's a preview feature included in .NET 6.

See the blog entry [HTTP/3 support in .NET 6](#).

Nullable Reference Type Annotations

Portions of the [ASP.NET Core 6.0 source code](#) has had [nullability annotations](#) applied.

By utilizing the new [Nullable feature in C# 8](#), ASP.NET Core can provide additional compile-time safety in the handling of reference types. For example, protecting against `null` reference exceptions. Projects that have opted in to using nullable annotations may see new build-time warnings from ASP.NET Core APIs.

To enable nullable reference types, add the following property to project files:

XML

```
<PropertyGroup>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

For more information, see [Nullable reference types](#).

Source Code Analysis

Several .NET compiler platform analyzers were added that inspect application code for problems such as incorrect middleware configuration or order, routing conflicts, etc. For more information, see [Code analysis in ASP.NET Core apps](#).

Web app template improvements

The web app templates:

- Use the new [minimal hosting model](#).
- Significantly reduces the number of files and lines of code required to create an app. For example, the ASP.NET Core empty web app creates one C# file with four lines of code and is a complete app.
- 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.

Template generated ports for Kestrel

Random ports are assigned during project creation for use by the Kestrel web server. Random ports help minimize a port conflict when multiple projects are run on the same machine.

When a project is created, a random HTTP port between 5000-5300 and a random HTTPS port between 7000-7300 is specified in the generated `Properties/launchSettings.json` file. The ports can be changed in the `Properties/launchSettings.json` file. If no port is specified, Kestrel defaults to the HTTP 5000 and HTTPS 5001 ports. For more information, see [Configure endpoints for the ASP.NET Core Kestrel web server](#).

New logging defaults

The following changes were made to both `appsettings.json` and `appsettings.Development.json`:

diff

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

The change from `"Microsoft": "Warning"` to `"Microsoft.AspNetCore": "Warning"` results in logging all informational messages from the `Microsoft` namespace *except* `Microsoft.AspNetCore`. For example, `Microsoft.EntityFrameworkCore` is now logged at the informational level.

Developer exception page Middleware added automatically

In the [development environment](#), the [DeveloperExceptionPageMiddleware](#) is added by default. It's no longer necessary to add the following code to web UI apps:

C#

```
if (app.Environment.IsDevelopment())  
{  
    app.UseDeveloperExceptionPage();  
}
```

Support for Latin1 encoded request headers in HttpSysServer

`HttpSysServer` now supports decoding request headers that are `Latin1` encoded by setting the [UseLatin1RequestHeaders](#) property on [HttpSysOptions](#) to `true`:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.UseHttpSys(o => o.UseLatin1RequestHeaders = true);

var app = builder.Build();

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

app.Run();
```

The ASP.NET Core Module logs include timestamps and PID

The [ASP.NET Core Module \(ANCM\) for IIS](#) (ANCM) enhanced diagnostic logs include timestamps and [PID](#) of the process emitting the logs. Logging timestamps and PID makes it easier to diagnose issues with overlapping process restarts in IIS when multiple IIS worker processes are running.

The resulting logs now resemble the sample output show below:

.NET CLI

```
[2021-07-28T19:23:44.076Z, PID: 11020] [aspnetcorev2.dll]
Initializing logs for 'C:\<path>\aspnetcorev2.dll'. Process Id:
11020. File Version: 16.0.21209.0. Description: IIS ASP.NET Core
Module V2. Commit: 96475a2acdf50d7599ba8e96583fa73efbe27912.
[2021-07-28T19:23:44.079Z, PID: 11020] [aspnetcorev2.dll] Resolving
hostfxr parameters for application: '.\InProcessWebSite.exe' argu-
ments: '' path: 'C:\Temp\e86ac4e9ced24bb6bacf1a9415e70753\'
[2021-07-28T19:23:44.080Z, PID: 11020] [aspnetcorev2.dll] Known dot-
net.exe location: ''
```

Configurable unconsumed incoming buffer size for IIS

The IIS server previously only buffered 64 KiB of unconsumed request bodies. The 64 KiB buffering resulted in reads being constrained to that maximum size, which impacts the performance with large incoming bodies such as uploads. In .NET 6, the default buffer size changes from 64 KiB to 1 MiB which should improve throughput for large uploads. In our tests, a 700 MiB upload that used to take 9 seconds now only takes 2.5 seconds.

The downside of a larger buffer size is an increased per-request memory consumption when the app isn't quickly reading from the request body. So, in addition to changing

the default buffer size, the buffer size configurable, allowing apps to configure the buffer size based on workload.

View Components Tag Helpers

Consider a view component with an optional parameter, as shown in the following code:

C#

```
class MyViewComponent
{
    IViewComponentResult Invoke(bool showSomething = false) { ... }
}
```

With ASP.NET Core 6, the tag helper can be invoked without having to specify a value for the `showSomething` parameter:

razor

```
<vc:my />
```

Angular template updated to Angular 12

The ASP.NET Core 6.0 template for Angular now uses [Angular 12](#).

The React template has been updated to [React 17](#).

Configurable buffer threshold before writing to disk in Json.NET output formatter

Note: We recommend using the [System.Text.Json](#) output formatter except when the `Newtonsoft.Json` serializer is required for compatibility reasons. The `System.Text.Json` serializer is fully `async` and works efficiently for larger payloads.

The `Newtonsoft.Json` output formatter by default buffers responses up to 32 KiB in memory before buffering to disk. This is to avoid performing synchronous IO, which can result in other side-effects such as thread starvation and application deadlocks. However, if the response is larger than 32 KiB, considerable disk I/O occurs. The memory threshold is now configurable via the [MvcNewtonsoftJsonOptions.OutputFormatterMemoryBufferThreshold](#) property before buffering to disk:

C#

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

builder.Services.AddRazorPages()
    .AddNewtonsoftJson(options =>
    {
        options.OutputFormatterMemoryBufferThreshold = 48 *
1024;
    });

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

For more information, see [this GitHub pull request](#) and the [Newtonsoft.Json.OutputFormatterTest.cs](#) file.

Faster get and set for HTTP headers

New APIs were added to expose all common headers available on [Microsoft.Net.Http.Headers.HeaderNames](#) as properties on the [IHeaderDictionary](#) resulting in an easier to use API. For example, the in-line middleware in the following code gets and sets both request and response headers using the new APIs:

C#

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

var app = builder.Build();

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

app.Use(async (context, next) =>
{
    var hostHeader = context.Request.Headers.Host;
    app.Logger.LogInformation("Host header: {host}", hostHeader);
    context.Response.Headers.XPoweredBy = "ASP.NET Core 6.0";
    await next.Invoke(context);
    var dateHeader = context.Response.Headers.Date;
    app.Logger.LogInformation("Response date: {date}", dateHeader);
});

app.Run();
```

For implemented headers the get and set accessors are implemented by going directly to the field and bypassing the lookup. For non-implemented headers, the accessors can bypass the initial lookup against implemented headers and directly perform the

`Dictionary<string, StringValues>` lookup. Avoiding the lookup results in faster access for both scenarios.

Async streaming

ASP.NET Core now supports asynchronous streaming from controller actions and responses from the JSON formatter. Returning an `IAsyncEnumerable` from an action no longer buffers the response content in memory before it gets sent. Not buffering helps reduce memory usage when returning large datasets that can be asynchronously enumerated.

Note that Entity Framework Core provides implementations of `IAsyncEnumerable` for querying the database. The improved support for `IAsyncEnumerable` in ASP.NET Core in .NET 6 can make using EF Core with ASP.NET Core more efficient. For example, the following code no longer buffers the product data into memory before sending the response:

C#

```
public IActionResult GetMovies()
{
    return Ok(_context.Movie);
}
```

However, when using lazy loading in EF Core, this new behavior may result in errors due to concurrent query execution while the data is being enumerated. Apps can revert back to the previous behavior by buffering the data:

C#

```
public async Task<IActionResult> GetMovies2()
{
    return Ok(await _context.Movie.ToListAsync());
}
```

See the related [announcement](#) for additional details about this change in behavior.

HTTP logging middleware

HTTP logging is a new built-in middleware that logs information about HTTP requests and HTTP responses including the headers and entire body:

C#


```

var builder = WebApplication.CreateBuilder(args);

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

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

app.Run();

```

Navigating to `/` with the previous code logs information similar to the following output:

.NET CLI

```

info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Protocol: HTTP/2
      Method: GET
      Scheme: https
      PathBase:
      Path: /
      Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
      Accept-Encoding: gzip, deflate, br
      Accept-Language: en-US,en;q=0.9
      Cache-Control: max-age=0
      Connection: close
      Cookie: [Redacted]
      Host: localhost:44372
      User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/95.0.4638.54
Safari/537.36 Edg/95.0.1020.30
      sec-ch-ua: [Redacted]
      sec-ch-ua-mobile: [Redacted]
      sec-ch-ua-platform: [Redacted]
      upgrade-insecure-requests: [Redacted]
      sec-fetch-site: [Redacted]
      sec-fetch-mode: [Redacted]
      sec-fetch-user: [Redacted]
      sec-fetch-dest: [Redacted]
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 200
      Content-Type: text/plain; charset=utf-8

```

The preceding output was enabled with the following `appsettings.Development.json` file:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware": "Information"
    }
  }
}
```

HTTP logging provides logs of:

- HTTP Request information
- Common properties
- Headers
- Body
- HTTP Response information

To configure the HTTP logging middleware, specify [HttpLoggingOptions](#):

C#

```
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddHttpLogging(logging =>
{
    // Customize HTTP logging.
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("My-Request-Header");
    logging.ResponseHeaders.Add("My-Response-Header");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
});

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

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

app.Run();
```

ICollectionSocketFeature

The [ICollectionSocketFeature](#) request feature provides access to the underlying accept socket associated with the current request. It can be accessed via the [FeatureCollection](#)

on `HttpContext`.

For example, the following app sets the `LingerState` property on the accepted socket:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureEndpointDefaults(listenOptions => listenOptions.Use((connection, next) =>
    {
        var socketFeature = connection.Features.Get<IConnectionSocketFeature>();
        socketFeature.Socket.LingerState = new LingerOption(true, seconds: 10);
        return next();
    }));
});
var app = builder.Build();
app.MapGet("/", (Func<string>)(() => "Hello world"));
await app.RunAsync();
```

Generic type constraints in Razor

When defining generic type parameters in Razor using the `@typeparam` directive, generic type constraints can now be specified using the standard C# syntax:

Smaller SignalR, Blazor Server, and MessagePack scripts

The SignalR, MessagePack, and Blazor Server scripts are now significantly smaller, enabling smaller downloads, less JavaScript parsing and compiling by the browser, and faster start-up. The size reductions:

- `signalr.js`: 70%
- `blazor.server.js`: 45%

The smaller scripts are a result of a community contribution from [Ben Adams](#). For more information on the details of the size reduction, see [Ben's GitHub pull request](#).

Enable Redis profiling sessions

A community contribution from [Gabriel Lucaci](#) enables Redis profiling session with [Microsoft.Extensions.Caching.StackExchangeRedis](#):

C#

```
using StackExchange.Redis.Profiling;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddStackExchangeRedisCache(options =>
{
    options.ProfilingSession = () => new ProfilingSession();
});
```

For more information, see [StackExchange.Redis Profiling](#).

Shadow copying in IIS

An experimental feature has been added to the [ASP.NET Core Module \(ANCM\) for IIS](#) to add support for [shadow copying application assemblies](#). Currently .NET locks application binaries when running on Windows making it impossible to replace binaries when the app is running. While our recommendation remains to use an [app offline file](#), we recognize there are certain scenarios (for example FTP deployments) where it isn't possible to do so.

In such scenarios, enable shadow copying by customizing the ASP.NET Core module handler settings. In most cases, ASP.NET Core apps do not have a `web.config` checked into source control that you can modify. In ASP.NET Core, `web.config` is ordinarily generated by the SDK. The following sample `web.config` can be used to get started:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <!-- To customize the asp.net core module uncomment and edit the
  following section.
  For more info see https://go.microsoft.com/fwlink/?linkid=838655 -->
  <system.webServer>
    <handlers>
      <remove name="aspNetCore"/>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCore-
ModuleV2" resourceType="Unspecified"/>
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%" argu-
ments="%LAUNCHER_ARGS%" stdoutLogEnabled="false" stdoutLog-
File=".\logs\stdout">
      <handlerSettings>
        <handlerSetting name="experimentalEnableShadowCopy" val-
ue="true" />
      </handlerSettings>
    </aspNetCore>
  </system.webServer>
</configuration>
```

```
<handlerSetting name="shadowCopyDirectory" value="../Shadow-
CopyDirectory/" />
<!-- Only enable handler logging if you encounter issues-->
<!--<handlerSetting name="debugFile" value=".\\logs\\aspnet-
core-debug.log" />-->
<!--<handlerSetting name="debugLevel" value="FILE,TRACE" />--
>
</handlerSettings>
</aspNetCore>
</system.webServer>
</configuration>
```

Shadow copying in IIS is an experimental feature that is not guaranteed to be part of ASP.NET Core. Please leave feedback on IIS Shadow copying in [this GitHub issue](#).

Additional resources

- [Code samples migrated to the new minimal hosting model in 6.0](#)
- [What's new in .NET 6](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)