

What's new in ASP.NET Core 3.0

Article • 09/28/2023

This article highlights the most significant changes in ASP.NET Core 3.0 with links to relevant documentation.

Blazor

Blazor is a new framework in ASP.NET Core for building interactive client-side web UI with .NET:

- Create rich interactive UIs using C#.
- Share server-side and client-side app logic written in .NET.
- Render the UI as HTML and CSS for wide browser support, including mobile browsers.

Blazor framework supported scenarios:

- Reusable UI components (Razor components)
- Client-side routing
- Component layouts
- Support for dependency injection
- Forms and validation
- Supply Razor components in Razor class libraries
- JavaScript interop

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

Blazor Server

Blazor decouples component rendering logic from how UI updates are applied. Blazor Server provides support for hosting Razor components on the server in an ASP.NET Core app. UI updates are handled over a SignalR connection. Blazor Server is supported in ASP.NET Core 3.0.

Blazor WebAssembly (Preview)

Blazor apps can also be run directly in the browser using a WebAssembly-based .NET runtime. Blazor WebAssembly is in preview and *not* supported in ASP.NET Core 3.0. Blazor WebAssembly will be supported in a future release of ASP.NET Core.

Razor components

Blazor apps are built from components. Components are self-contained chunks of user interface (UI), such as a page, dialog, or form. Components are normal .NET classes that define UI rendering logic and client-side event handlers. You can create rich interactive web apps without JavaScript.




Components in Blazor are typically authored using Razor syntax, a natural blend of HTML and C#. Razor components are similar to Razor Pages and MVC views in that they both use Razor. Unlike pages and views, which are based on a request-response model, components are used specifically for handling UI composition.

gRPC

[gRPC](#) :

- Is a popular, high-performance RPC (remote procedure call) framework.
- Offers an opinionated contract-first approach to API development.
- Uses modern technologies such as:
 - HTTP/2 for transport.
 - Protocol Buffers as the interface description language.
 - Binary serialization format.
- Provides features such as:
 - Authentication
 - Bidirectional streaming and flow control.
 - Cancellation and timeouts.

gRPC functionality in ASP.NET Core 3.0 includes:

- [Grpc.AspNetCore](#) : An ASP.NET Core framework for hosting gRPC services. gRPC on ASP.NET Core integrates with standard ASP.NET Core features like logging, dependency injection (DI), authentication, and authorization.
- [Grpc.Net.Client](#) : A gRPC client for .NET Core that builds upon the familiar `HttpClient`.
- [Grpc.Net.ClientFactory](#) : gRPC client integration with `HttpClientFactory`.

For more information, see [Overview for gRPC on .NET](#).

SignalR

See [Update SignalR code](#) for migration instructions. SignalR now uses `System.Text.Json` to serialize/deserialize JSON messages. See [Switch to Newtonsoft.Json](#) for instructions to restore the `Newtonsoft.Json`-based serializer.

In the JavaScript and .NET Clients for SignalR, support was added for automatic reconnection. By default, the client tries to reconnect immediately and retry after 2, 10, and 30 seconds if necessary. If the client successfully reconnects, it receives a new connection ID. Automatic reconnect is opt-in:

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect()
    .build();
```

The reconnection intervals can be specified by passing an array of millisecond-based durations:

JavaScript

```
.withAutomaticReconnect([0, 3000, 5000, 10000, 15000, 30000])
//.withAutomaticReconnect([0, 2000, 10000, 30000]) The default
intervals.
```

A custom implementation can be passed in for full control of the reconnection intervals.

If the reconnection fails after the last reconnect interval:

- The client considers the connection is offline.
- The client stops trying to reconnect.

During reconnection attempts, update the app UI to notify the user that the reconnection is being attempted.

To provide UI feedback when the connection is interrupted, the SignalR client API has been expanded to include the following event handlers:

- `onreconnecting`: Gives developers an opportunity to disable UI or to let users know the app is offline.
- `onreconnected`: Gives developers an opportunity to update the UI once the connection is reestablished.

The following code uses `onreconnecting` to update the UI while trying to connect:

JavaScript

```
connection.onreconnecting((error) => {
    const status = `Connection lost due to error "${error}".
    Reconnecting.`;
    document.getElementById("messageInput").disabled = true;
    document.getElementById("sendButton").disabled = true;
    document.getElementById("connectionStatus").innerText = status;
});
```

The following code uses `onreconnected` to update the UI on connection:

JavaScript

```
connection.onreconnected((connectionId) => {
    const status = `Connection reestablished. Connected.`;
    document.getElementById("messageInput").disabled = false;
    document.getElementById("sendButton").disabled = false;
    document.getElementById("connectionStatus").innerText = status;
});
```

SignalR 3.0 and later provides a custom resource to authorization handlers when a hub method requires authorization. The resource is an instance of `HubInvocationContext`. The `HubInvocationContext` includes the:

- `HubCallerContext`
- Name of the hub method being invoked.
- Arguments to the hub method.

Consider the following example of a chat room app allowing multiple organization sign-in via Azure Active Directory. Anyone with a Microsoft account can sign in to chat, but only members of the owning organization can ban users or view users' chat histories. The app could restrict certain functionality from specific users.

C#

```
public class DomainRestrictedRequirement :
    AuthorizationHandler<DomainRestrictedRequirement,
    HubInvocationContext>,
    IAuthorizationRequirement
{
    protected override Task
    HandleRequirementAsync(AuthorizationHandlerContext context,
        DomainRestrictedRequirement requirement,
        HubInvocationContext resource)
    {
        if (context.User?.Identity?.Name == null)
        {
```

```

        return Task.CompletedTask;
    }

    if (IsUserAllowedToDoThis(resource.HubMethodName, context-
t.User.Identity.Name))
    {
        context.Succeed(requirement);
    }

    return Task.CompletedTask;
}

private bool IsUserAllowedToDoThis(string hubMethodName, string
currentUsername)
{
    if (hubMethodName.Equals("banUser",
StringComparison.OrdinalIgnoreCase))
    {
        return currentUsername.Equals("bob42@jabbr.net",
StringComparison.OrdinalIgnoreCase);
    }

    return currentUsername.EndsWith("@jabbr.net",
StringComparison.OrdinalIgnoreCase));
}
}

```

In the preceding code, `DomainRestrictedRequirement` serves as a custom `IAuthorizationRequirement`. Because the `HubInvocationContext` resource parameter is being passed in, the internal logic can:

- Inspect the context in which the Hub is being called.
- Make decisions on allowing the user to execute individual Hub methods.

Individual Hub methods can be marked with the name of the policy the code checks at run-time. As clients attempt to call individual Hub methods, the

`DomainRestrictedRequirement` handler runs and controls access to the methods.

Based on the way the `DomainRestrictedRequirement` controls access:

- All logged-in users can call the `SendMessage` method.
- Only users who have logged in with a `@jabbr.net` email address can view users' histories.
- Only `bob42@jabbr.net` can ban users from the chat room.

C#

```

[Authorize]
public class ChatHub : Hub
{

```

```

    public void SendMessage(string message)
    {
    }

    [Authorize("DomainRestricted")]
    public void BanUser(string username)
    {
    }

    [Authorize("DomainRestricted")]
    public void ViewUserHistory(string username)
    {
    }
}

```

Creating the `DomainRestricted` policy might involve:

- In `Startup.cs`, adding the new policy.
- Provide the custom `DomainRestrictedRequirement` requirement as a parameter.
- Registering `DomainRestricted` with the authorization middleware.

C#

```

services
    .AddAuthorization(options =>
    {
        options.AddPolicy("DomainRestricted", policy =>
        {
            policy.Requirements.Add(new
DomainRestrictedRequirement());
        });
    });

```

SignalR hubs use [Endpoint Routing](#). SignalR hub connection was previously done explicitly:

C#

```

app.UseSignalR(routes =>
{
    routes.MapHub<ChatHub>("hubs/chat");
});

```

In the previous version, developers needed to wire up controllers, Razor pages, and hubs in a variety of places. Explicit connection results in a series of nearly-identical routing segments:

C#

```
app.UseSignalR(routes =>
{
    routes.MapHub<ChatHub>("hubs/chat");
});

app.UseRouting(routes =>
{
    routes.MapRazorPages();
});
```

SignalR 3.0 hubs can be routed via endpoint routing. With endpoint routing, typically all routing can be configured in `UseRouting`:

```
C#

app.UseRouting(routes =>
{
    routes.MapRazorPages();
    routes.MapHub<ChatHub>("hubs/chat");
});
```

ASP.NET Core 3.0 SignalR added:

Client-to-server streaming. With client-to-server streaming, server-side methods can take instances of either an `IAsyncEnumerable<T>` or `ChannelReader<T>`. In the following C# sample, the `UploadStream` method on the Hub will receive a stream of strings from the client:

```
C#

public async Task UploadStream(IAsyncEnumerable<string> stream)
{
    await foreach (var item in stream)
    {
        // process content
    }
}
```

.NET client apps can pass either an `IAsyncEnumerable<T>` or `ChannelReader<T>` instance as the `stream` argument of the `UploadStream` Hub method above.

After the `for` loop has completed and the local function exits, the stream completion is sent:

```
C#
```

```

async IEnumerable<string> clientStreamData()
{
    for (var i = 0; i < 5; i++)
    {
        var data = await FetchSomeData();
        yield return data;
    }
}

await connection.SendAsync("UploadStream", clientStreamData());

```

JavaScript client apps use the SignalR `Subject` (or an [RxJS Subject](#)) for the `stream` argument of the `UploadStream` Hub method above.

JavaScript

```

let subject = new signalR.Subject();
await connection.send("StartStream", "MyAsciiArtStream", subject);

```

The JavaScript code could use the `subject.next` method to handle strings as they are captured and ready to be sent to the server.

JavaScript

```

subject.next("example");
subject.complete();

```

Using code like the two preceding snippets, real-time streaming experiences can be created.

New JSON serialization

ASP.NET Core 3.0 now uses [System.Text.Json](#) by default for JSON serialization:

- Reads and writes JSON asynchronously.
- Is optimized for UTF-8 text.
- Typically higher performance than `Newtonsoft.Json`.

To add `Json.NET` to ASP.NET Core 3.0, see [Add Newtonsoft.Json-based JSON format support](#).

New Razor directives

The following list contains new Razor directives:

- **@attribute**: The `@attribute` directive applies the given attribute to the class of the generated page or view. For example, `@attribute [Authorize]`.
- **@implements**: The `@implements` directive implements an interface for the generated class. For example, `@implements IDisposable`.

IdentityServer4 supports authentication and authorization for web APIs and SPAs

ASP.NET Core 3.0 offers authentication in Single Page Apps (SPAs) using the support for web API authorization. ASP.NET Core Identity for authenticating and storing users is combined with [IdentityServer4](#) for implementing OpenID Connect.

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core 3.0. It enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see [the IdentityServer4 documentation](#) or [Authentication and authorization for SPAs](#).

Certificate and Kerberos authentication

Certificate authentication requires:

- Configuring the server to accept certificates.
- Adding the authentication middleware in `Startup.Configure`.
- Adding the certificate authentication service in `Startup.ConfigureServices`.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
        .AddCertificate();
    // Other service configuration removed.
}
```

```
public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
{
    app.UseAuthentication();
    // Other app configuration removed.
}
```

Options for certificate authentication include the ability to:

- Accept self-signed certificates.
- Check for certificate revocation.
- Check that the proffered certificate has the right usage flags in it.

A default user principal is constructed from the certificate properties. The user principal contains an event that enables supplementing or replacing the principal. For more information, see [Configure certificate authentication in ASP.NET Core](#).

[Windows Authentication](#) has been extended onto Linux and macOS. In previous versions, Windows Authentication was limited to [IIS](#) and [HTTP.sys](#). In ASP.NET Core 3.0, [Kestrel](#) has the ability to use Negotiate, [Kerberos](#), and [NTLM on Windows](#), Linux, and macOS for Windows domain-joined hosts. Kestrel support of these authentication schemes is provided by the [Microsoft.AspNetCore.Authentication.Negotiate NuGet](#) [package](#). As with the other authentication services, configure authentication app wide, then configure the service:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(NegotiateDefaults.Authentication-
Scheme)
        .AddNegotiate();
    // Other service configuration removed.
}

public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
{
    app.UseAuthentication();
    // Other app configuration removed.
}
```

Host requirements:

- Windows hosts must have [Service Principal Names](#) (SPNs) added to the user account hosting the app.
- Linux and macOS machines must be joined to the domain.

- SPNs must be created for the web process.
- [Keytab files](#) must be generated and configured on the host machine.

For more information, see [Configure Windows Authentication in ASP.NET Core](#).

Template changes

The web UI templates (Razor Pages, MVC with controller and views) have the following removed:

- The cookie consent UI is no longer included. To enable the cookie consent feature in an ASP.NET Core 3.0 template-generated app, see [General Data Protection Regulation \(GDPR\) support in ASP.NET Core](#).
- Scripts and related static assets are now referenced as local files instead of using CDNs. For more information, see [Scripts and related static assets are now referenced as local files instead of using CDNs based on the current environment \(dotnet/AspNetCore.Docs #14350\)](#) [↗](#).

The Angular template updated to use Angular 8.

The Razor class library (RCL) template defaults to Razor component development by default. A new template option in Visual Studio provides template support for pages and views. When creating an RCL from the template in a command shell, pass the `--support-pages-and-views` option (`dotnet new razorclasslib --support-pages-and-views`).

Generic Host

The ASP.NET Core 3.0 templates use [.NET Generic Host in ASP.NET Core](#). Previous versions used [WebHostBuilder](#). Using the .NET Core Generic Host ([HostBuilder](#)) provides better integration of ASP.NET Core apps with other server scenarios that aren't web-specific. For more information, see [HostBuilder replaces WebHostBuilder](#).

Host configuration

Prior to the release of ASP.NET Core 3.0, environment variables prefixed with `ASPNETCORE_` were loaded for host configuration of the Web Host. In 3.0, `AddEnvironmentVariables` is used to load environment variables prefixed with `DOTNET_` for host configuration with `CreateDefaultBuilder`.

Changes to Startup constructor injection

The Generic Host only supports the following types for `Startup` constructor injection:

- [IHostEnvironment](#)
- `IWebHostEnvironment`
- [IConfiguration](#)

All services can still be injected directly as arguments to the `Startup.Configure` method. For more information, see [Generic Host restricts Startup constructor injection \(aspnet/Announcements #353\)](#) [↗](#).

Kestrel

- Kestrel configuration has been updated for the migration to the Generic Host. In 3.0, Kestrel is configured on the web host builder provided by `ConfigureWebHostDefaults`.
- Connection Adapters have been removed from Kestrel and replaced with Connection Middleware, which is similar to HTTP Middleware in the ASP.NET Core pipeline but for lower-level connections.
- The Kestrel transport layer has been exposed as a public interface in `Connections.Abstractions`.
- Ambiguity between headers and trailers has been resolved by moving trailing headers to a new collection.
- Synchronous I/O APIs, such as `HttpRequest.Body.Read`, are a common source of thread starvation leading to app crashes. In 3.0, `AllowSynchronousIO` is disabled by default.

For more information, see [Migrate from ASP.NET Core 2.2 to 3.0](#).

HTTP/2 enabled by default

HTTP/2 is enabled by default in Kestrel for HTTPS endpoints. HTTP/2 support for IIS or HTTP.sys is enabled when supported by the operating system.

EventCounters on request

The Hosting EventSource, `Microsoft.AspNetCore.Hosting`, emits the following new [EventCounter](#) types related to incoming requests:

- `requests-per-second`
- `total-requests`
- `current-requests`
- `failed-requests`

Endpoint routing

Endpoint Routing, which allows frameworks (for example, MVC) to work well with middleware, is enhanced:

- The order of middleware and endpoints is configurable in the request processing pipeline of `Startup.Configure`.
- Endpoints and middleware compose well with other ASP.NET Core-based technologies, such as Health Checks.
- Endpoints can implement a policy, such as CORS or authorization, in both middleware and MVC.
- Filters and attributes can be placed on methods in controllers.

For more information, see [Routing in ASP.NET Core](#).

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 the following articles:

- [Migrate from ASP.NET Core 2.2 to 3.0](#)
- [Health checks in ASP.NET Core](#)

Pipes on HttpContext

It's now possible to read the request body and write the response body using the [System.IO.Pipelines](#) API. The `HttpRequest.BodyReader` property provides a [PipeReader](#) that can be used to read the request body. The `HttpResponse.BodyWriter` property provides a [PipeWriter](#) that can be used to write the response body.

`HttpRequest.BodyReader` is an analogue of the `HttpRequest.Body` stream.

`HttpResponse.BodyWriter` is an analogue of the `HttpResponse.Body` stream.


Improved error reporting in IIS

Startup errors when hosting ASP.NET Core apps in IIS now produce richer diagnostic data. These errors are reported to the Windows Event Log with stack traces wherever applicable. In addition, all warnings, errors, and unhandled exceptions are logged to the Windows Event Log.

Worker Service and Worker SDK

.NET Core 3.0 introduces the new Worker Service app template. This template provides a starting point for writing long running services in .NET Core.

For more information, see:

- [.NET Core Workers as Windows Services](#) 
- [Background tasks with hosted services in ASP.NET Core](#)
- [Host ASP.NET Core in a Windows Service](#)

Forwarded Headers Middleware improvements

In previous versions of ASP.NET Core, calling [UseHsts](#) and [UseHttpsRedirection](#) were problematic when deployed to an Azure Linux or behind any reverse proxy other than IIS. The fix for previous versions is documented in [Forward the scheme for Linux and non-IIS reverse proxies](#).

This scenario is fixed in ASP.NET Core 3.0. The host enables the [Forwarded Headers Middleware](#) when the `ASPNETCORE_FORWARDEDHEADERS_ENABLED` environment variable is set to `true`. `ASPNETCORE_FORWARDEDHEADERS_ENABLED` is set to `true` in our container images.

Performance improvements

ASP.NET Core 3.0 includes many improvements that reduce memory usage and improve throughput:

- Reduction in memory usage when using the built-in dependency injection container for scoped services.
- Reduction in allocations across the framework, including middleware scenarios and routing.
- Reduction in memory usage for WebSocket connections.
- Memory reduction and throughput improvements for HTTPS connections.
- New optimized and fully asynchronous JSON serializer.
- Reduction in memory usage and throughput improvements in form parsing.

ASP.NET Core 3.0 only runs on .NET Core 3.0

As of ASP.NET Core 3.0, .NET Framework is no longer a supported target framework. Projects targeting .NET Framework can continue in a fully supported fashion using the [.NET Core 2.1 LTS release](#). Most ASP.NET Core 2.1.x related packages will be supported indefinitely, beyond the three-year LTS period for .NET Core 2.1.

For migration information, see [Port your code from .NET Framework to .NET Core](#).

Use the ASP.NET Core shared framework

The ASP.NET Core 3.0 shared framework, contained in the [Microsoft.AspNetCore.App metapackage](#), no longer requires an explicit `<PackageReference />` element in the project file. The shared framework is automatically referenced when using the `Microsoft.NET.Sdk.Web` SDK in the project file:

XML

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

Assemblies removed from the ASP.NET Core shared framework

The most notable assemblies removed from the ASP.NET Core 3.0 shared framework are:

- [Newtonsoft.Json](#) [↗] (Json.NET). To add Json.NET to ASP.NET Core 3.0, see [Add Newtonsoft.Json-based JSON format support](#). ASP.NET Core 3.0 introduces `System.Text.Json` for reading and writing JSON. For more information, see [New JSON serialization](#) in this document.
- [Entity Framework Core](#)

For a complete list of assemblies removed from the shared framework, see [Assemblies being removed from Microsoft.AspNetCore.App 3.0](#) [↗]. For more information on the motivation for this change, see [Breaking changes to Microsoft.AspNetCore.App in 3.0](#) [↗] and [A first look at changes coming in ASP.NET Core 3.0](#) [↗].

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](#)