What's new in ASP.NET Core 7.0

Article • 11/02/2023

This article highlights the most significant changes in ASP.NET Core 7.0 with links to relevant documentation.

Rate limiting middleware in ASP.NET Core

The `Microsoft.AspNetCore.RateLimiting` middleware provides rate limiting middleware. Apps configure rate limiting policies and then attach the policies to endpoints. For more information, see Rate limiting middleware in ASP.NET Core.

Authentication uses single scheme as DefaultScheme

As part of the work to simplify authentication, when there's only a single authentication scheme registered, it's automatically used as the DefaultScheme and doesn't need to be specified. For more information, see DefaultScheme.

MVC and Razor pages

Support for nullable models in MVC views and Razor Pages

Nullable page or view models are supported to improve the experience when using null state checking with ASP.NET Core apps:

```
C#

@model Product?
```

Bind with `IParsable<T>.TryParse` in MVC and API Controllers

The IParsable<TSelf>.TryParse API supports binding controller action parameter values. For more information, see Bind with IParsable<T>.TryParse.

Customize the cookie consent value

In ASP.NET Core versions earlier than 7, the cookie consent validation uses the cookie value `yes` to indicate consent. Now you can specify the value that represents consent. For example, you could use `true` instead of `yes`:

C#

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

builder.Services.AddRazorPages();
builder.Services.Configure<CookiePolicyOptions>(options =>
{
    options.CheckConsentNeeded = context => true;
    options.MinimumSameSitePolicy = SameSiteMode.None;
    options.ConsentCookieValue = "true";
});

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

For more information, see Customize the cookie consent value.

API controllers

Parameter binding with DI in API controllers

Parameter binding for API controller actions binds parameters through dependency injection when the type is configured as a service. This means it's no longer required to explicitly apply the [FromServices] attribute to a parameter. In the following code, both actions return the time:

C#

```
[Route("[controller]")]
[ApiController]
public class MyController : ControllerBase
{
    public ActionResult GetWithAttribute([FromServices] IDateTime
dateTime)
                                                          =>
Ok(dateTime.Now);

    [Route("noAttribute")]
    public ActionResult Get(IDateTime dateTime) => Ok(dateTime.Now);
}
```

In rare cases, automatic DI can break apps that have a type in DI that is also accepted in an API controllers action method. It's not common to have a type in DI and as an argument in an API controller action. To disable automatic binding of parameters, set DisableImplicitFromServicesParameters

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddSingleton<IDateTime, SystemDateTime>();

builder.Services.Configure<ApiBehaviorOptions>(options =>
{
    options.DisableImplicitFromServicesParameters = true;
});

var app = builder.Build();

app.MapControllers();

app.Run();
```

In ASP.NET Core 7.0, types in DI are checked at app startup with IServiceProviderIsService to determine if an argument in an API controller action comes from DI or from the other sources.

The new mechanism to infer binding source of API Controller action parameters uses the following rules:

1. A previously specified BindingInfo.BindingSource is never overwritten.
2. A complex type parameter, registered in the DI container, is assigned BindingSource.Services.
3. A complex type parameter, not registered in the DI container, is assigned BindingSource.Body.
4. A parameter with a name that appears as a route value in ***any*** route template is assigned BindingSource.Path.
5. All other parameters are BindingSource.Query.

JSON property names in validation errors

By default, when a validation error occurs, model validation produces a ModelStateDictionary with the property name as the error key. Some apps, such as single page apps, benefit from using JSON property names for validation errors

generated from Web APIs. The following code configures validation to use the SystemTextJsonValidationMetadataProvider to use JSON property names:

C#

```
using Microsoft.AspNetCore.Mvc.ModelBinding.Metadata;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.ModelMetadataDetailsProviders.Add(new
SystemTextJsonValidationMetadataProvider());
});

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The following code configures validation to use the NewtonsoftJsonValidationMetadataProvider to use JSON property name when using Json.NET:

C#

```
using Microsoft.AspNetCore.Mvc.NewtonsoftJson;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.ModelMetadataDetailsProviders.Add(new
NewtonsoftJsonValidationMetadataProvider());
}).AddNewtonsoftJson();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

For more information, see Use JSON property names in validation errors

Minimal APIs

Filters in Minimal API apps

Minimal API filters allow developers to implement business logic that supports:

- Running code before and after the route handler.
- Inspecting and modifying parameters provided during a route handler invocation.
- Intercepting the response behavior of a route handler.

Filters can be helpful in the following scenarios:

- Validating the request parameters and body that are sent to an endpoint.
- Logging information about the request and response.
- Validating that a request is targeting a supported API version.

For more information, see Filters in Minimal API apps

Bind arrays and string values from headers and query strings

In ASP.NET 7, binding query strings to an array of primitive types, string arrays, and StringValues is supported:

C#

```
// Bind query string values to a primitive type array.
// GET  /tags?q=1&q=2&q=3
app.MapGet("/tags", (int[] q) =>
                      $"tag1: {q[0]} , tag2: {q[1]}, tag3: {q[2]}");

// Bind to a string array.
// GET /tags2?names=john&names=jack&names=jane
app.MapGet("/tags2", (string[] names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3:
{names[2]}");

// Bind to StringValues.
// GET /tags3?names=john&names=jack&names=jane
app.MapGet("/tags3", (StringValues names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3:
{names[2]}");
```

Binding query strings or header values to an array of complex types is supported when the type has `TryParse` implemented. For more information, see Bind arrays and string values from headers and query strings.

For more information, see Add endpoint summary or description.

Bind the request body as a `Stream` or `PipeReader`

The request body can bind as a Stream or PipeReader to efficiently support scenarios where the user has to process data and:

- Store the data to blob storage or enqueue the data to a queue provider.
- Process the stored data with a worker process or cloud function.

For example, the data might be enqueued to Azure Queue storage or stored in Azure Blob storage.

For more information, see Bind the request body as a Stream or PipeReader

New Results.Stream overloads

We introduced new Results.Stream overloads to accommodate scenarios that need access to the underlying HTTP response stream without buffering. These overloads also improve cases where an API streams data to the HTTP response stream, like from Azure Blob Storage. The following example uses ImageSharp to return a reduced size of the specified image:

C#

```
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Formats.Jpeg;
using SixLabors.ImageSharp.Processing;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/process-image/{strImage}", (string strImage, HttpContext
http, CancellationToken token) =>
{
    http.Response.Headers.CacheControl = $"public,max-age={TimeS-
pan.FromHours(24).TotalSeconds}";
    return Results.Stream(stream => ResizeImageAsync(strImage,
stream, token), "image/jpeg");
});

async Task ResizeImageAsync(string strImage, Stream stream,
```

```
CancellationToken token)
{
    var strPath = $"wwwroot/img/{strImage}";
    using var image = await Image.LoadAsync(strPath, token);
    int width = image.Width / 2;
    int height = image.Height / 2;
    image.Mutate(x =>x.Resize(width, height));
    await image.SaveAsync(stream, JpegFormat.Instance, cancellation-
Token: token);
}
```

For more information, see Stream examples

Typed results for minimal APIs

In .NET 6, the IResult interface was introduced to represent values returned from minimal APIs that don't utilize the implicit support for JSON serializing the returned object to the HTTP response. The static Results class is used to create varying `IResult` objects that represent different types of responses. For example, setting the response status code or redirecting to another URL. The `IResult` implementing framework types returned from these methods were internal however, making it difficult to verify the specific `IResult` type being returned from methods in a unit test.

In .NET 7 the types implementing `IResult` are public, allowing for type assertions when testing. For example:

C#

```
[TestClass()]
public class WeatherApiTests
{
    [TestMethod()]
    public void MapWeatherApiTest()
    {
        var result = WeatherApi.GetAllWeathers();
        Assert.IsInstanceOfType(result, type-
of(Ok<WeatherForecast[]>));
    }
}
```

Improved unit testability for minimal route handlers

IResult implementation types are now publicly available in the Microsoft.AspNetCore.Http.HttpResults namespace. The `IResult` implementation types

can be used to unit test minimal route handlers when using named methods instead of lambdas.

The following code uses the Ok<TValue> class:

C#

```
[Fact]
public async Task GetTodoReturnsTodoFromDatabase()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    context.Todos.Add(new Todo
    {
        Id = 1,
        Title = "Test title",
        Description = "Test description",
        IsDone = false
    });

    await context.SaveChangesAsync();

    // Act
    var result = await TodoEndpointsV1.GetTodo(1, context);

    //Assert
    Assert.IsType<Results<Ok<Todo>, NotFound>>(result);

    var okResult = (Ok<Todo>)result.Result;

    Assert.NotNull(okResult.Value);
    Assert.Equal(1, okResult.Value.Id);
}
```

For more information, see IResult implementation types.

New HttpResult interfaces

The following interfaces in the Microsoft.AspNetCore.Http namespace provide a way to detect the `IResult` type at runtime, which is a common pattern in filter implementations:

- IContentTypeHttpResult
- IFileHttpResult
- INestedHttpResult
- IStatusCodeHttpResult
- IValueHttpResult

- IValueHttpResult<TValue>

For more information, see IHttpResult interfaces.

OpenAPI improvements for minimal APIs

`Microsoft.AspNetCore.OpenApi` NuGet package

The Microsoft.AspNetCore.OpenApi package allows interactions with OpenAPI specifications for endpoints. The package acts as a link between the OpenAPI models that are defined in the `Microsoft.AspNetCore.OpenApi` package and the endpoints that are defined in Minimal APIs. The package provides an API that examines an endpoint's parameters, responses, and metadata to construct an OpenAPI annotation type that is used to describe an endpoint.

C#

```
app.MapPost("/todoitems/{id}", async (int id, Todo todo, TodoDb db)
=>
{
    todo.Id = id;
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
})
.WithOpenApi();
```

Call `WithOpenApi` with parameters

The WithOpenApi method accepts a function that can be used to modify the OpenAPI annotation. For example, in the following code, a description is added to the first parameter of the endpoint:

C#

```
app.MapPost("/todo2/{id}", async (int id, Todo todo, TodoDb db) =>
{
    todo.Id = id;
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
})
```

```
.WithOpenApi(generatedOperation =>
{
    var parameter = generatedOperation.Parameters[0];
    parameter.Description = "The ID associated with the created
Todo";
    return generatedOperation;
});
```

Provide endpoint descriptions and summaries

Minimal APIs now support annotating operations with descriptions and summaries for OpenAPI spec generation. You can call extension methods WithDescription and WithSummary or use attributes [EndpointDescription] and [EndpointSummary]).

For more information, see OpenAPI in minimal API apps

File uploads using IFormFile and IFormFileCollection

Minimal APIs now support file upload with `IFormFile` and `IFormFileCollection`. The following code uses IFormFile and IFormFileCollection to upload file:

C#

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

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

app.MapPost("/upload", async (IFormFile file) =>
{
    var tempFile = Path.GetTempFileName();
    app.Logger.LogInformation(tempFile);
    using var stream = File.OpenWrite(tempFile);
    await file.CopyToAsync(stream);
});

app.MapPost("/upload_many", async (IFormFileCollection myFiles) =>
{
    foreach (var file in myFiles)
    {
        var tempFile = Path.GetTempFileName();
        app.Logger.LogInformation(tempFile);
        using var stream = File.OpenWrite(tempFile);
        await file.CopyToAsync(stream);
    }
});

app.Run();
```

Authenticated file upload requests are supported using an Authorization header, a client certificate, or a cookie header.

There is no built-in support for antiforgery. However, it can be implemented using the IAntiforgery service.

`[AsParameters]` attribute enables parameter binding for argument lists

The [AsParameters] attribute enables parameter binding for argument lists. For more information, see Parameter binding for argument lists with [AsParameters].

Minimal APIs and API controllers

New problem details service

The problem details service implements the IProblemDetailsService interface, which supports creating Problem Details for HTTP APIs.

For more information, see Problem details service.

Route groups

The MapGroup extension method helps organize groups of endpoints with a common prefix. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata which add endpoint metadata.

For example, the following code creates two similar groups of endpoints:

C#

```
app.MapGroup("/public/todos")
    .MapTodosApi()
    .WithTags("Public");

app.MapGroup("/private/todos")
    .MapTodosApi()
    .WithTags("Private")
    .AddEndpointFilterFactory(QueryPrivateTodos)
    .RequireAuthorization();


EndpointFilterDelegate QueryPrivateTodos(EndpointFilterFactoryContext
```

```
factoryContext, EndpointFilterDelegate next)
{
    var dbContextIndex = -1;

    foreach (var argument in
factoryContext.MethodInfo.GetParameters())
    {
        if (argument.ParameterType == typeof(TodoDb))
        {
            dbContextIndex = argument.Position;
            break;
        }
    }

    // Skip filter if the method doesn't have a TodoDb parameter.
    if (dbContextIndex < 0)
    {
        return next;
    }

    return async invocationContext =>
    {
        var dbContext = invocationContext.GetArgument<TodoDb>(dbCon-
textIndex);
        dbContext.IsPrivate = true;

        try
        {
            return await next(invocationContext);
        }
        finally
        {
            // This should only be relevant if you're pooling or oth-
erwise reusing the DbContext instance.
            dbContext.IsPrivate = false;
        }
    };
}
```

C#

```
public static RouteGroupBuilder MapTodosApi(this RouteGroupBuilder
group)
{
    group.MapGet("/", GetAllTodos);
    group.MapGet("/{id}", GetTodo);
    group.MapPost("/", CreateTodo);
    group.MapPut("/{id}", UpdateTodo);
    group.MapDelete("/{id}", DeleteTodo);

    return group;
}
```

In this scenario, you can use a relative address for the `Location` header in the `201 Created` result:

C#

```
public static async Task<Created<Todo>> CreateTodo(Todo todo, TodoDb
database)
{
    await database.AddAsync(todo);
    await database.SaveChangesAsync();

    return TypedResults.Created($"{todo.Id}", todo);
}
```

The first group of endpoints will only match requests prefixed with `/public/todos` and are accessible without any authentication. The second group of endpoints will only match requests prefixed with `/private/todos` and require authentication.

The `QueryPrivateTodos` endpoint filter factory is a local function that modifies the route handler's `TodoDb` parameters to allow to access and store private todo data.

Route groups also support nested groups and complex prefix patterns with route parameters and constraints. In the following example, and route handler mapped to the `user` group can capture the `{org}` and `{group}` route parameters defined in the outer group prefixes.

The prefix can also be empty. This can be useful for adding endpoint metadata or filters to a group of endpoints without changing the route pattern.

C#

```
var all = app.MapGroup("").WithOpenApi();
var org = all.MapGroup("{org}");
var user = org.MapGroup("{user}");
user.MapGet("", (string org, string user) => $"{org}/{user}");
```

Adding filters or metadata to a group behaves the same way as adding them individually to each endpoint before adding any extra filters or metadata that may have been added to an inner group or specific endpoint.

C#

```
var outer = app.MapGroup("/outer");
var inner = outer.MapGroup("/inner");

inner.AddEndpointFilter((context, next) =>
```

```
{
    app.Logger.LogInformation("/inner group filter");
    return next(context);
});

outer.AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("/outer group filter");
    return next(context);
});

inner.MapGet("/", () => "Hi!").AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("MapGet filter");
    return next(context);
});
```

In the above example, the outer filter will log the incoming request before the inner filter even though it was added second. Because the filters were applied to different groups, the order they were added relative to each other does not matter. The order filters are added does matter if applied to the same group or specific endpoint.

A request to `/outer/inner/` will log the following:

.NET CLI

```
/outer group filter
/inner group filter
MapGet filter
```

gRPC

JSON transcoding

gRPC JSON transcoding is an extension for ASP.NET Core that creates RESTful JSON APIs for gRPC services. gRPC JSON transcoding allows:

- Apps to call gRPC services with familiar HTTP concepts.
- ASP.NET Core gRPC apps to support both gRPC and RESTful JSON APIs without replicating functionality.
- Experimental support for generating OpenAPI from transcoded RESTful APIs by integrating with Swashbuckle.

For more information, see gRPC JSON transcoding in ASP.NET Core gRPC apps and Use OpenAPI with gRPC JSON transcoding ASP.NET Core apps.

gRPC health checks in ASP.NET Core

The gRPC health checking protocol is a standard for reporting the health of gRPC server apps. An app exposes health checks as a gRPC service. They are typically used with an external monitoring service to check the status of an app.

gRPC ASP.NET Core has added built-in support for gRPC health checks with the Grpc.AspNetCore.HealthChecks package. Results from .NET health checks are reported to callers.

For more information, see gRPC health checks in ASP.NET Core.

Improved call credentials support

Call credentials are the recommended way to configure a gRPC client to send an auth token to the server. gRPC clients support two new features to make call credentials easier to use:

- Support for call credentials with plaintext connections. Previously, a gRPC call only sent call credentials if the connection was secured with TLS. A new setting on `GrpcChannelOptions`, called `UnsafeUseInsecureChannelCallCredentials`, allows this behavior to be customized. There are security implications to not securing a connection with TLS.
- A new method called `AddCallCredentials` is available with the gRPC client factory. `AddCallCredentials` is a quick way to configure call credentials for a gRPC client and integrates well with dependency injection (DI).

The following code configures the gRPC client factory to send `Authorization` metadata:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
       o.Address = new Uri("https://localhost:5001");
    })
    .AddCallCredentials((context, metadata) =>
    {
       if (!string.IsNullOrEmpty(_token))
       {
          metadata.Add("Authorization", $"Bearer {_token}");
       }
       return Task.CompletedTask;
    });
```

For more information, see Configure a bearer token with the gRPC client factory.

SignalR

Client results

The server now supports requesting a result from a client. This requires the server to use `ISingleClientProxy.InvokeAsync` and the client to return a result from its `.On` handler. Strongly-typed hubs can also return values from interface methods.

For more information, see Client results

Dependency injection for SignalR hub methods

SignalR hub methods now support injecting services through dependency injection (DI).

Hub constructors can accept services from DI as parameters, which can be stored in properties on the class for use in a hub method. For more information, see Inject services into a hub

Blazor

Handle location changing events and navigation state

In .NET 7, Blazor supports location changing events and maintaining navigation state. This allows you to warn users about unsaved work or to perform related actions when the user performs a page navigation.

For more information, see the following sections of the *Routing and navigation* article:

- Navigation options
- Handle/prevent location changes

Empty Blazor project templates

Blazor has two new project templates for starting from a blank slate. The new **Blazor Server App Empty** and **Blazor WebAssembly App Empty** project templates are just like their non-empty counterparts but without example code. These empty templates only include a basic home page, and we've removed Bootstrap so that you can start with a different CSS framework.

For more information, see the following articles:

- Tooling for ASP.NET Core Blazor
- ASP.NET Core Blazor project structure

Blazor custom elements

The Microsoft.AspNetCore.Components.CustomElements package enables building standards based custom DOM elements using Blazor.

For more information, see ASP.NET Core Razor components.

Bind modifiers (`@bind:after`, `@bind:get`, `@bind:set`)

> **Important**
>
> The `@bind:after`/`@bind:get`/`@bind:set` features are receiving further updates at this time. To take advantage of the latest updates, confirm that you've installed the latest SDK.
>
> Using an event callback parameter (`[Parameter] public EventCallback<string> ValueChanged { get; set; }`) isn't supported. Instead, pass an Action-returning or Task-returning method to `@bind:set`/`@bind:after`.
>
> For more information, see the following resources:
>
> - Blazor @bind:after not working on .NET 7 RTM release (dotnet/aspnetcore #44957)
> - BindGetSetAfter701 sample app (javiercn/BindGetSetAfter701 GitHub repository)

In .NET 7, you can run asynchronous logic after a binding event has completed using the new `@bind:after` modifier. In the following example, the `PerformSearch` asynchronous method runs automatically after any changes to the search text are detected:

razor

```
<input @bind="searchText" @bind:after="PerformSearch" />

@code {
    private string searchText;
```

```
    private async Task PerformSearch()
    {
        ...
    }
}
```

In .NET 7, it's also easier to set up binding for component parameters. Components can support two-way data binding by defining a pair of parameters:

- `@bind:get`: Specifies the value to bind.
- `@bind:set`: Specifies a callback for when the value changes.

The `@bind:get` and `@bind:set` modifiers are always used together.

Examples:

razor

```
@* Elements *@

<input type="text" @bind="text" @bind:after="() => { }" />

<input type="text" @bind:get="text" @bind:set="(value) => { }" />

<input type="text" @bind="text" @bind:after="AfterAsync" />

<input type="text" @bind:get="text" @bind:set="SetAsync" />

<input type="text" @bind="text" @bind:after="() => { }" />

<input type="text" @bind:get="text" @bind:set="(value) => { }" />

<input type="text" @bind="text" @bind:after="AfterAsync" />

<input type="text" @bind:get="text" @bind:set="SetAsync" />

@* Components *@

<InputText @bind-Value="text" @bind-Value:after="() => { }" />

<InputText @bind-Value:get="text" @bind-Value:set="(value) => { }" />

<InputText @bind-Value="text" @bind-Value:after="AfterAsync" />

<InputText @bind-Value:get="text" @bind-Value:set="SetAsync" />

<InputText @bind-Value="text" @bind-Value:after="() => { }" />

<InputText @bind-Value:get="text" @bind-Value:set="(value) => { }" />

<InputText @bind-Value="text" @bind-Value:after="AfterAsync" />
```

```
<InputText @bind-Value:get="text" @bind-Value:set="SetAsync" />

@code {
    private string text = "";

    private void After(){}
    private void Set() {}
    private Task AfterAsync() { return Task.CompletedTask; }
    private Task SetAsync(string value) { return Task.CompletedTask;
}
}
```

For more information on the `InputText` component, see ASP.NET Core Blazor input components.

Hot Reload improvements

In .NET 7, Hot Reload support includes the following:

- Components reset their parameters to their default values when a value is removed.
- Blazor WebAssembly:
  - Add new types.
  - Add nested classes.
  - Add static and instance methods to existing types.
  - Add static fields and methods to existing types.
  - Add static lambdas to existing methods.
  - Add lambdas that capture `this` to existing methods that already captured `this` previously.

Dynamic authentication requests with MSAL in Blazor WebAssembly

New in .NET 7, Blazor WebAssembly supports creating dynamic authentication requests at runtime with custom parameters to handle advanced authentication scenarios.

For more information, see the following articles:

- Secure ASP.NET Core Blazor WebAssembly
- ASP.NET Core Blazor WebAssembly additional security scenarios

Blazor WebAssembly debugging improvements

Blazor WebAssembly debugging has the following improvements:

- Support for the **Just My Code** setting to show or hide type members that aren't from user code.
- Support for inspecting multidimensional arrays.
- **Call Stack** now shows the correct name for asynchronous methods.
- Improved expression evaluation.
- Correct handling of the `new` keyword on derived members.
- Support for debugger-related attributes in `System.Diagnostics`.

`System.Security.Cryptography` support on WebAssembly

.NET 6 supported the SHA family of hashing algorithms when running on WebAssembly. .NET 7 enables more cryptographic algorithms by taking advantage of SubtleCrypto, when possible, and falling back to a .NET implementation when SubtleCrypto can't be used. The following algorithms are supported on WebAssembly in .NET 7:

- SHA1
- SHA256
- SHA384
- SHA512
- HMACSHA1
- HMACSHA256
- HMACSHA384
- HMACSHA512
- AES-CBC
- PBKDF2
- HKDF

For more information, see Developers targeting browser-wasm can use Web Crypto APIs (dotnet/runtime #40074).

Inject services into custom validation attributes

You can now inject services into custom validation attributes. Blazor sets up the `ValidationContext` so that it can be used as a service provider.

For more information, see ASP.NET Core Blazor forms validation.

`Input*` components outside of an `EditContext`/`EditForm`

The built-in input components are now supported outside of a form in Razor component markup.

For more information, see ASP.NET Core Blazor input components.

Project template changes

When .NET 6 was released last year, the HTML markup of the `_Host` page (`Pages/_Host.chstml`) was split between the `_Host` page and a new `_Layout` page (`Pages/_Layout.chstml`) in the .NET 6 Blazor Server project template.

In .NET 7, the HTML markup has been recombined with the `_Host` page in project templates.

Several additional changes were made to the Blazor project templates. It isn't feasible to list every change to the templates in the documentation. To migrate an app to .NET 7 in order to adopt all of the changes, see Migrate from ASP.NET Core 6.0 to 7.0.

Experimental `QuickGrid` component

The new `QuickGrid` component provides a convenient data grid component for most common requirements and as a reference architecture and performance baseline for anyone building Blazor data grid components.

For more information, see ASP.NET Core Blazor QuickGrid component.

Live demo: QuickGrid for Blazor sample app

Virtualization enhancements

Virtualization enhancements in .NET 7:

- The `Virtualize` component supports using the document itself as the scroll root, as an alternative to having some other element with `overflow-y: scroll` applied.
- If the `Virtualize` component is placed inside an element that requires a specific child tag name, `SpacerElement` allows you to obtain or set the virtualization spacer tag name.

For more information, see the following sections of the *Virtualization* article:

- Root-level virtualization
- Control the spacer element tag name

`MouseEventArgs` updates

`MovementX` and `MovementY` have been added to `MouseEventArgs`.

For more information, see ASP.NET Core Blazor event handling.

New Blazor loading page

The Blazor WebAssembly project template has a new loading UI that shows the progress of loading the app.

For more information, see ASP.NET Core Blazor startup.

Improved diagnostics for authentication in Blazor WebAssembly

To help diagnose authentication issues in Blazor WebAssembly apps, detailed logging is available.

For more information, see ASP.NET Core Blazor logging.

JavaScript interop on WebAssembly

JavaScript `[JSImport]`/`[JSExport]` interop API is a new low-level mechanism for using .NET in Blazor WebAssembly and JavaScript-based apps. With this new JavaScript interop capability, you can invoke .NET code from JavaScript using the .NET WebAssembly runtime and call into JavaScript functionality from .NET without any dependency on the Blazor UI component model.

For more information:

- JavaScript JSImport/JSExport interop with ASP.NET Core Blazor: Pertains only to Blazor WebAssembly apps.
- Run .NET from JavaScript: Pertains only to JavaScript apps that don't depend on the Blazor UI component model.

Conditional registration of the authentication state provider

Prior to the release of .NET 7, `AuthenticationStateProvider` was registered in the service container with `AddScoped`. This made it difficult to debug apps, as it forced a specific order of service registrations when providing a custom implementation. Due to internal framework changes over time, it's no longer necessary to register `AuthenticationStateProvider` with `AddScoped`.

In developer code, make the following change to the authentication state provider service registration:

diff

```
- builder.Services.AddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
+ builder.Services.TryAddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
```

In the preceding example, `ExternalAuthStateProvider` is the developer's service implementation.

Improvements to the .NET WebAssembly build tools

New features in the `wasm-tools` workload for .NET 7 that help improve performance and handle exceptions:

- WebAssembly Single Instruction, Multiple Data (SIMD) support (only with AOT, not supported by Apple Safari)
- WebAssembly exception handling support

For more information, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

Blazor Hybrid

External URLs

An option has been added that permits opening external webpages in the browser.

For more information, see ASP.NET Core Blazor Hybrid routing and navigation.

Security

New guidance is available for Blazor Hybrid security scenarios. For more information, see the following articles:

- ASP.NET Core Blazor Hybrid authentication and authorization
- ASP.NET Core Blazor Hybrid security considerations

Performance

Output caching middleware

Output caching is a new middleware that stores responses from a web app and serves them from a cache rather than computing them every time. Output caching differs from response caching in the following ways:

- The caching behavior is configurable on the server.
- Cache entries can be programmatically invalidated.
- Resource locking mitigates the risk of cache stampede and thundering herd.
- Cache revalidation means the server can return a `304 Not Modified` HTTP status code instead of a cached response body.
- The cache storage medium is extensible.

For more information, see Overview of caching and Output caching middleware.

HTTP/3 improvements

This release:

- Makes HTTP/3 fully supported by ASP.NET Core, it's no longer experimental.
- Improves Kestrel's support for HTTP/3. The two main areas of improvement are feature parity with HTTP/1.1 and HTTP/2, and performance.
- Provides full support for UseHttps(ListenOptions, X509Certificate2) with HTTP/3. Kestrel offers advanced options for configuring connection certificates, such as hooking into Server Name Indication (SNI).
- Adds support for HTTP/3 on HTTP.sys and IIS.

The following example shows how to use an SNI callback to resolve TLS options:

C#

```
using Microsoft.AspNetCore.Server.Kestrel.Core;
using Microsoft.AspNetCore.Server.Kestrel.Https;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;
```

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(options =>
{
    options.ListenAnyIP(8080, listenOptions =>
    {
        listenOptions.Protocols =
HttpProtocols.Http1AndHttp2AndHttp3;
        listenOptions.UseHttps(new TlsHandshakeCallbackOptions
        {
            OnConnection = context =>
            {
                var options = new SslServerAuthenticationOptions
                {
                    ServerCertificate =

MyResolveCertForHost(context.ClientHelloInfo.ServerName)
                };
                return new ValueTask<SslServerAuthenticationOptions>
(options);
            },
        });
    });
});
```

Significant work was done in .NET 7 to reduce HTTP/3 allocations. You can see some of those improvements in the following GitHub PR's:

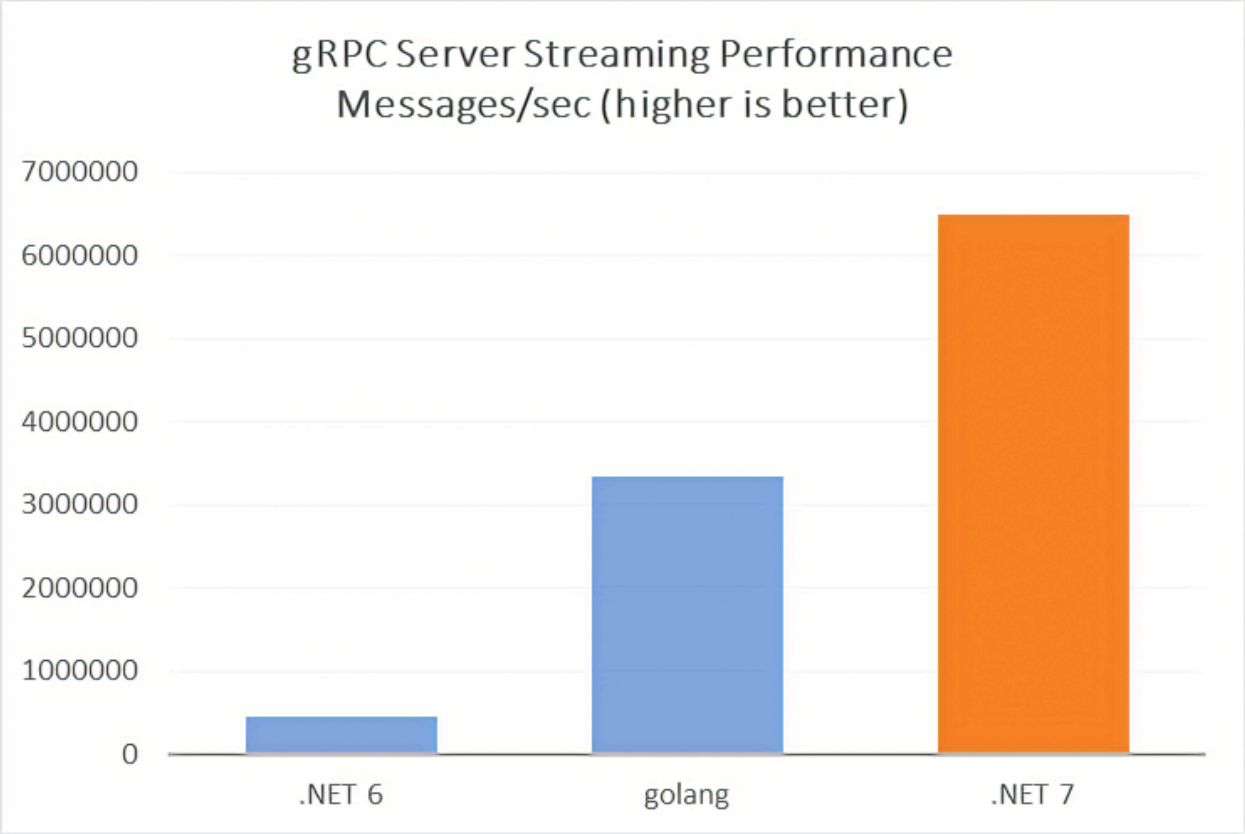
- HTTP/3: Avoid per-request cancellation token allocations
- HTTP/3: Avoid ConnectionAbortedException allocations
- HTTP/3: ValueTask pooling

HTTP/2 Performance improvements

.NET 7 introduces a significant re-architecture of how Kestrel processes HTTP/2 requests. ASP.NET Core apps with busy HTTP/2 connections will experience reduced CPU usage and higher throughput.

Previously, the HTTP/2 multiplexing implementation relied on a lock controlling which request can write to the underlying TCP connection. A thread-safe queue replaces the write lock. Now, rather than fighting over which thread gets to use the write lock, requests now queue up and a dedicated consumer processes them. Previously wasted CPU resources are available to the rest of the app.

One place where these improvements can be noticed is in gRPC, a popular RPC framework that uses HTTP/2. Kestrel + gRPC benchmarks show a dramatic improvement:



Changes were made in the HTTP/2 frame writing code that improves performance when there are multiple streams trying to write data on a single HTTP/2 connection. We now dispatch TLS work to the thread pool and more quickly release a write lock that other streams can acquire to write their data. The reduction in wait times can yield significant performance improvements in cases where there is contention for this write lock. A gRPC benchmark with 70 streams on a single connection (with TLS) showed a ~15% improvement in requests per second (RPS) with this change.

Http/2 WebSockets support

.NET 7 introduces Websockets over HTTP/2 support for Kestrel, the SignalR JavaScript client, and SignalR with Blazor WebAssembly.

Using WebSockets over HTTP/2 takes advantage of new features such as:

- Header compression.
- Multiplexing, which reduces the time and resources needed when making multiple requests to the server.

These supported features are available in Kestrel on all HTTP/2 enabled platforms. The version negotiation is automatic in browsers and Kestrel, so no new APIs are needed.

For more information, see Http/2 WebSockets support.

Kestrel performance improvements on high core machines

Kestrel uses ConcurrentQueue<T> for many purposes. One purpose is scheduling I/O operations in Kestrel's default Socket transport. Partitioning the `ConcurrentQueue` based on the associated socket reduces contention and increases throughput on machines with many CPU cores.

Profiling on high core machines on .NET 6 showed significant contention in one of Kestrel's other `ConcurrentQueue` instances, the `PinnedMemoryPool` that Kestrel uses to cache byte buffers.

In .NET 7, Kestrel's memory pool is partitioned the same way as its I/O queue, which leads to much lower contention and higher throughput on high core machines. On the 80 core ARM64 VMs, we're seeing over 500% improvement in responses per second (RPS) in the TechEmpower plaintext benchmark. On 48 Core AMD VMs, the improvement is nearly 100% in our HTTPS JSON benchmark.

`ServerReady` event to measure startup time

Apps using EventSource can measure the startup time to understand and optimize startup performance. The new ServerReady event in Microsoft.AspNetCore.Hosting represents the point where the server is ready to respond to requests.

Server

New ServerReady event for measuring startup time

The ServerReady event has been added to measure startup time of ASP.NET Core apps.

IIS

Shadow copying in IIS

Shadow copying app assemblies to the ASP.NET Core Module (ANCM) for IIS can provide a better end user experience than stopping the app by deploying an app offline file.

For more information, see Shadow copying in IIS.

Miscellaneous

Kestrel full certificate chain improvements

HttpsConnectionAdapterOptions has a new ServerCertificateChain property of type X509Certificate2Collection, which makes it easier to validate certificate chains by allowing a full chain including intermediate certificates to be specified. See dotnet/aspnetcore#21513 for more details.

dotnet watch

Improved console output for dotnet watch

The console output from dotnet watch has been improved to better align with the logging of ASP.NET Core and to stand out with 😮 emojis😍 .

Here's an example of what the new output looks like:

```
dotnet watch 🔥 Hot reload enabled. For a list of supported edits, see https://aka.ms/dotnet/hot-reload.
  💡 Press "Ctrl + R" to restart.
dotnet watch 🔧 Building...
  Determining projects to restore...
  All projects are up-to-date for restore.
  WebApplication18 -> C:\tmp\WebApplication18\bin\Debug\net6.0\WebApplication18.dll
dotnet watch 🚀 Started
dotnet watch 🕵 Files changed: .\Pages\Index.cshtml, .\Pages\Index.cshtml.cs
dotnet watch 🔥 Hot reload of changes succeeded.
```

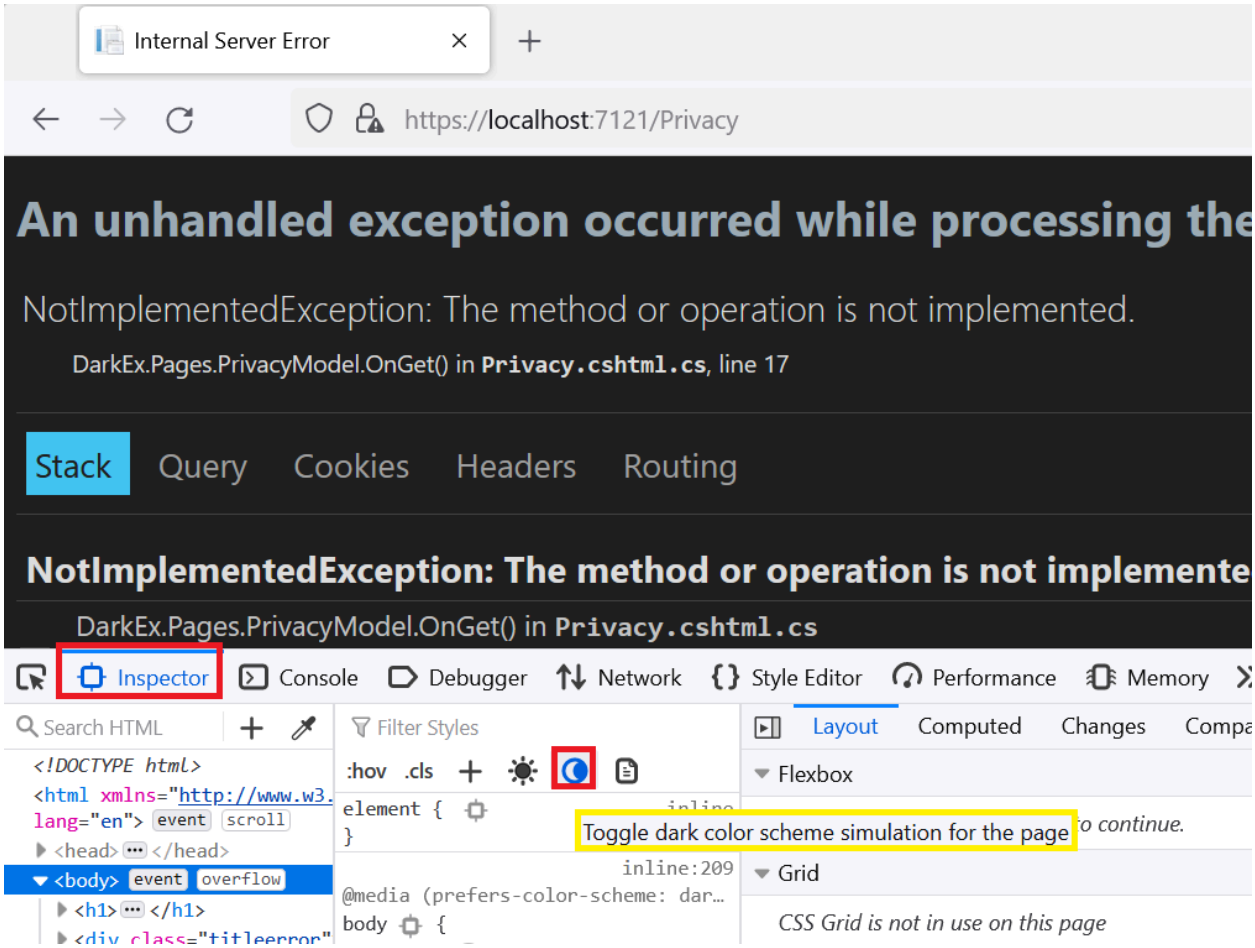
For more information, see this GitHub pull request.

Configure dotnet watch to always restart for rude edits

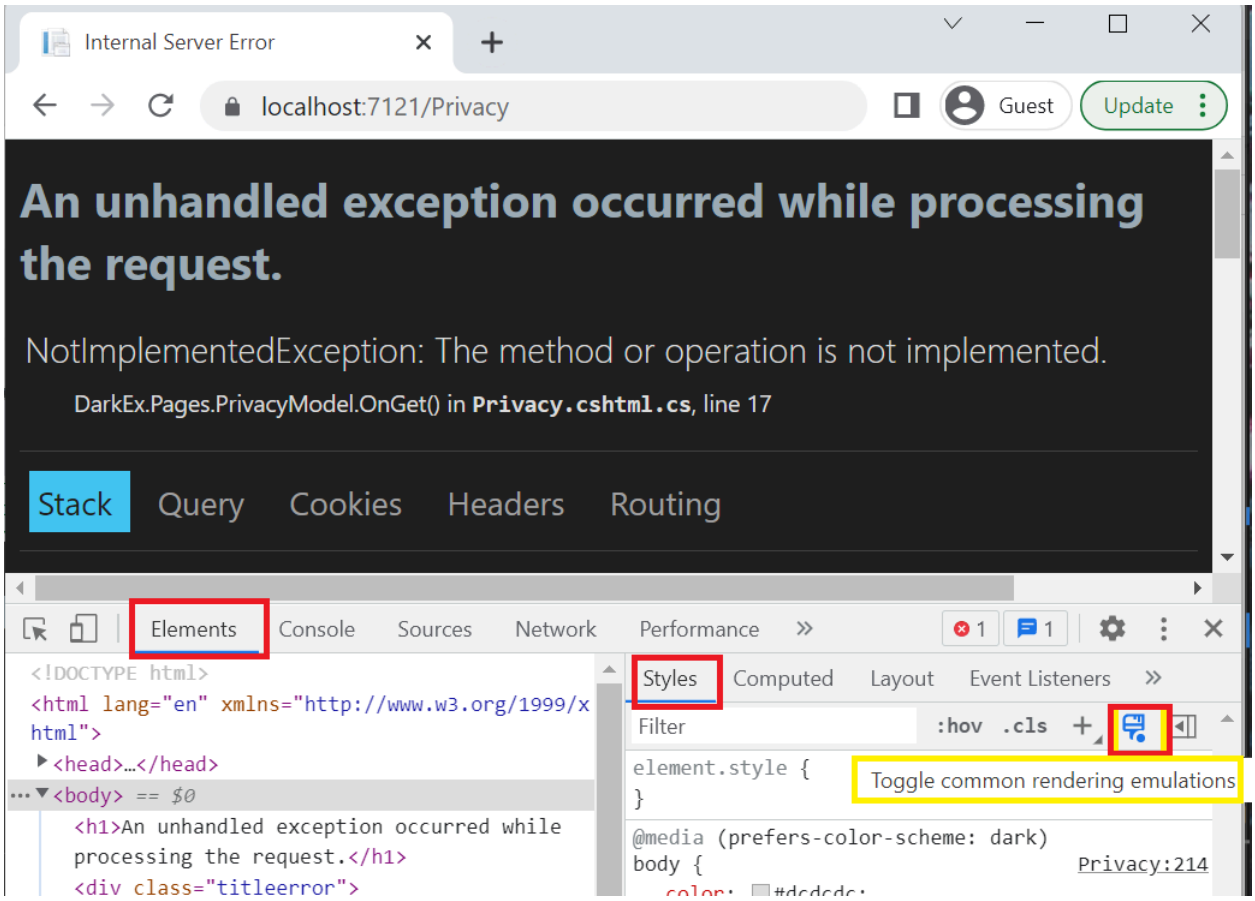
Rude edits are edits that can't be hot reloaded. To configure dotnet watch to always restart without a prompt for rude edits, set the `DOTNET_WATCH_RESTART_ON_RUDE_EDIT` environment variable to `true`.

Developer exception page dark mode

Dark mode support has been added to the developer exception page, thanks to a contribution by Patrick Westerhoff. To test dark mode in a browser, from the developer tools page, set the mode to dark. For example, in Firefox:


Internal Server Error
https://localhost:7121/Privacy
An unhandled exception occurred while processing the
NotImplementedException: The method or operation is not implemented.
DarkEx.Pages.PrivacyModel.OnGet() in Privacy.cshtml.cs, line 17
Stack
Query
Cookies
Headers
Routing
NotImplementedException: The method or operation is not implemente
DarkEx.Pages.PrivacyModel.OnGet() in Privacy.cshtml.cs
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Search HTML
Filter Styles
Layout
Computed
Changes
Flexbox
Toggle dark color scheme simulation for the page
Grid
CSS Grid is not in use on this page

In Chrome:


Internal Server Error
localhost:7121/Privacy
Guest
Update
An unhandled exception occurred while processing the request.
NotImplementedException: The method or operation is not implemented.
DarkEx.Pages.PrivacyModel.OnGet() in Privacy.cshtml.cs, line 17
Stack
Query
Cookies
Headers
Routing
Elements
Console
Sources
Network
Performance
Styles
Computed
Layout
Event Listeners
Filter
:hov .cls
element.style {
}
Toggle common rendering emulations
@media (prefers-color-scheme: dark)
body {
Privacy:214

Project template option to use Program.Main method instead of top-level statements

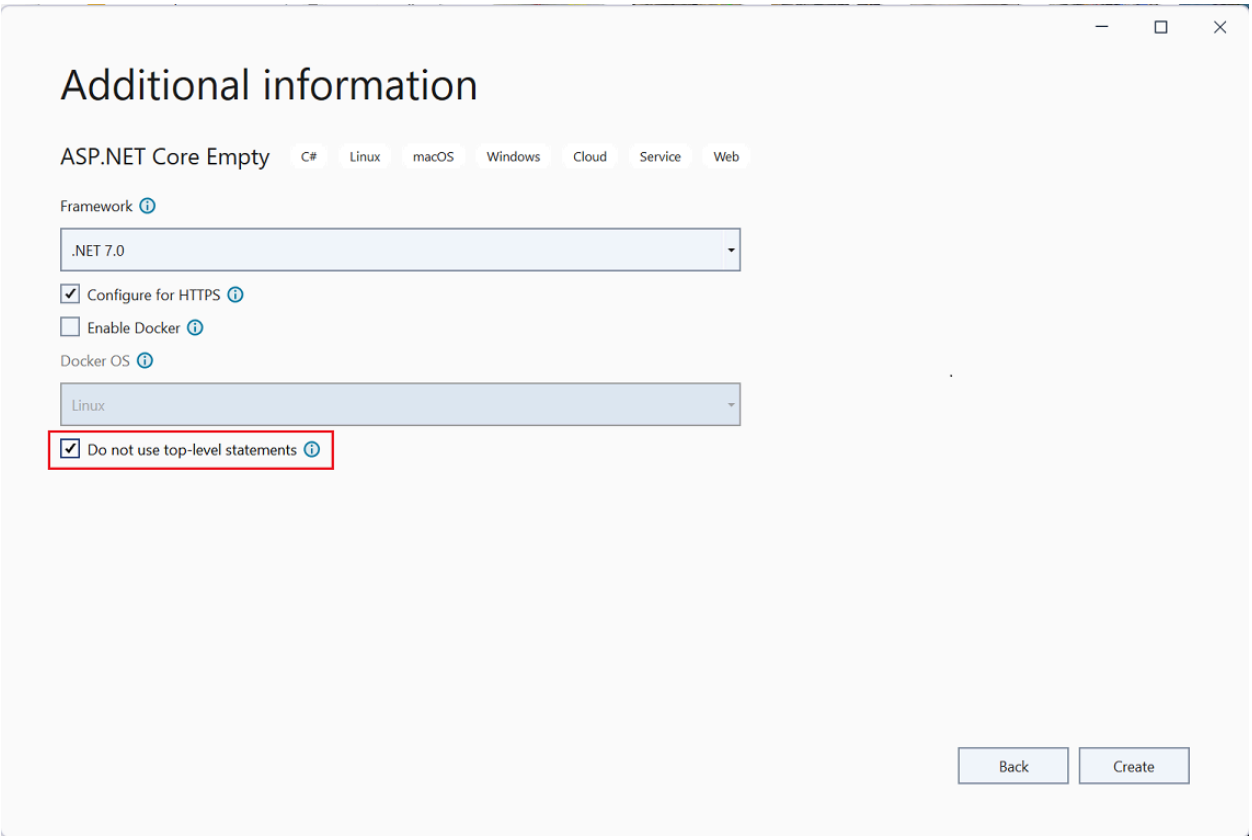
The .NET 7 templates include an option to not use top-level statements and generate a `namespace` and a `Main` method declared on a `Program` class.

Using the .NET CLI, use the `--use-program-main` option:

.NET CLI

```
dotnet new web --use-program-main
```

With Visual Studio, select the new **Do not use top-level statements** checkbox during project creation:



Updated Angular and React templates

The Angular project template has been updated to Angular 14. The React project template has been updated to React 18.2.

Manage JSON Web Tokens in development with dotnet user-jwts

The new `dotnet user-jwts` command line tool can create and manage app specific local JSON Web Tokens (JWTs). For more information, see Manage JSON Web Tokens in development with dotnet user-jwts.

Support for additional request headers in W3CLogger

You can now specify additional request headers to log when using the W3C logger by calling `AdditionalRequestHeaders()` on W3CLoggerOptions:

C#

```
services.AddW3CLogging(logging =>
{
    logging.AdditionalRequestHeaders.Add("x-forwarded-for");
    logging.AdditionalRequestHeaders.Add("x-client-ssl-protocol");
});
```

For more information, see W3CLogger options.

Request decompression

The new Request decompression middleware:

- Enables API endpoints to accept requests with compressed content.
- Uses the Content-Encoding HTTP header to automatically identify and decompress requests which contain compressed content.
- Eliminates the need to write code to handle compressed requests.

For more information, see Request decompression middleware.