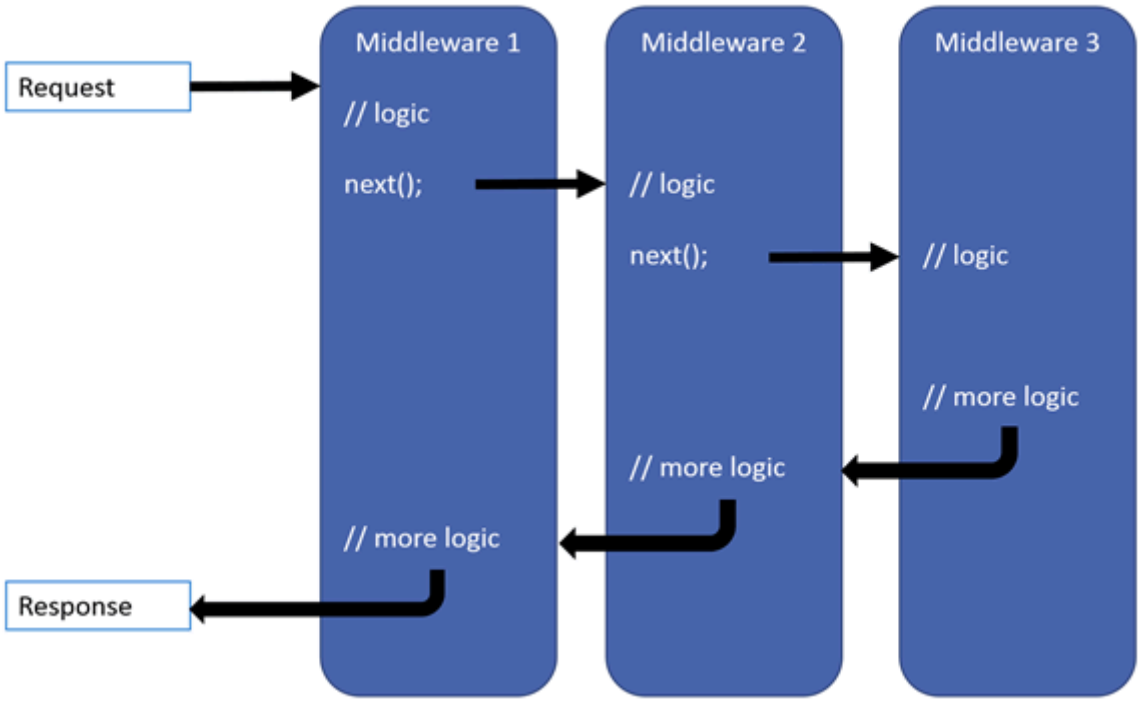
Create a middleware pipeline with `WebApplication`

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other. The following diagram demonstrates the concept. The thread of execution follows the black arrows.



Each delegate can perform operations before and after the next delegate. Exception-handling delegates should be called early in the pipeline, so they can catch exceptions that occur in later stages of the pipeline.

The simplest possible ASP.NET Core app sets up a single request delegate that handles all requests. This case doesn't include an actual request pipeline. Instead, a single anonymous function is called in response to every HTTP request.

C#

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

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello world!");
});

app.Run();
```

Chain multiple request delegates together with Use. The `next` parameter represents the next delegate in the pipeline. You can short-circuit the pipeline by *not* calling the `next` parameter. You can typically perform actions both before and after the `next` delegate, as the following example demonstrates:

C#

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

app.Use(async (context, next) =>
{
    // Do work that can write to the Response.
    await next.Invoke();
    // Do logging or other work that doesn't write to the Response.
});

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from 2nd delegate.");
});

app.Run();
```

Short-circuiting the request pipeline

When a delegate doesn't pass a request to the next delegate, it's called *short-circuiting the request pipeline*. Short-circuiting is often desirable because it avoids unnecessary work. For example, Static File Middleware can act as a *terminal middleware* by processing a request for a static file and short-circuiting the rest of the pipeline. Middleware added to the pipeline before the middleware that terminates further processing still processes code after their `next.Invoke` statements. However, see the following warning about attempting to write to a response that has already been sent.

> ⚠ **Warning**
>
> Don't call `next.Invoke` during or after the response has been sent to the client. After an HttpResponse has started, changes result in an exception. For example, setting headers and a status code throw an exception after the response starts. Writing to the response body after calling `next`:
>
> - May cause a protocol violation, such as writing more than the stated `Content-Length`.
> - May corrupt the body format, such as writing an HTML footer to a CSS file.

HasStarted is a useful hint to indicate if headers have been sent or the body has been written to.

For more information, see Short-circuit middleware after routing.

`Run` delegates

Run delegates don't receive a `next` parameter. The first `Run` delegate is always terminal and terminates the pipeline. `Run` is a convention. Some middleware components may expose `Run[Middleware]` methods that run at the end of the pipeline:

C#

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

app.Use(async (context, next) =>
{
    // Do work that can write to the Response.
    await next.Invoke();
    // Do logging or other work that doesn't write to the Response.
});

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from 2nd delegate.");
});

app.Run();
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

In the preceding example, the `Run` delegate writes `"Hello from 2nd delegate."` to the response and then terminates the pipeline. If another `Use` or `Run` delegate is added after the `Run` delegate, it's not called.

Prefer app.Use overload that requires passing the context to next

The non-allocating app.Use extension method:

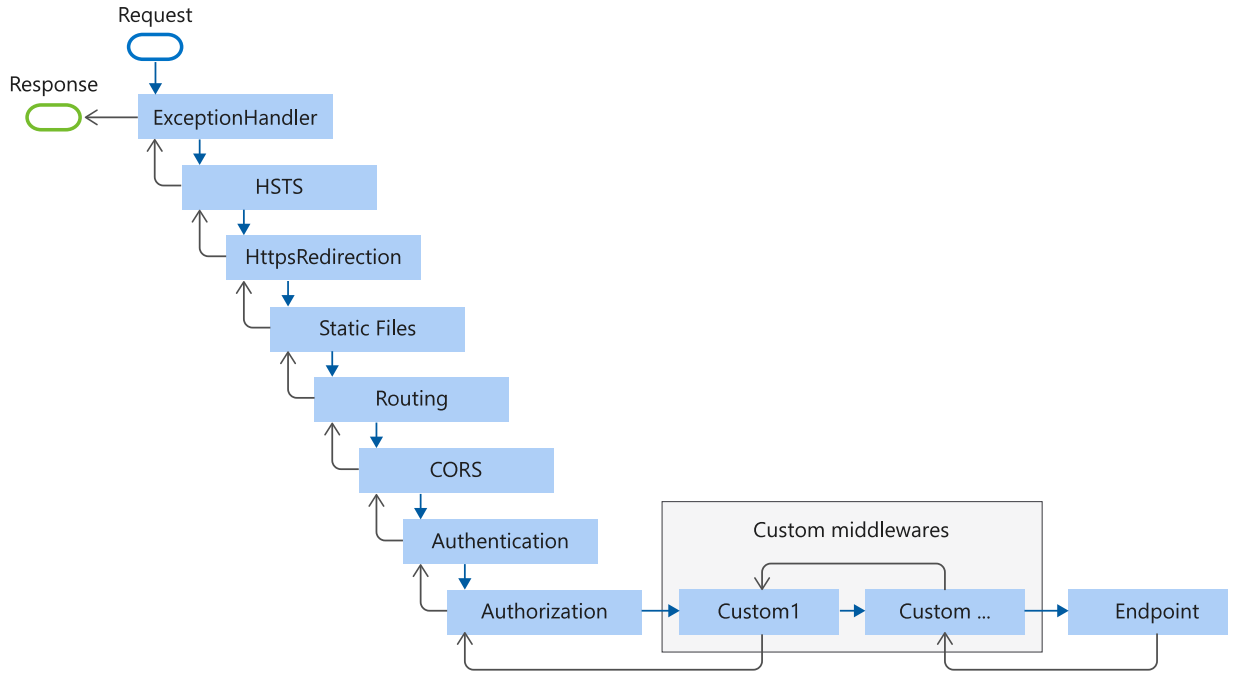
- Requires passing the context to `next`.

- Saves two internal per-request allocations that are required when using the other overload.

For more information, see this GitHub issue.

Middleware order

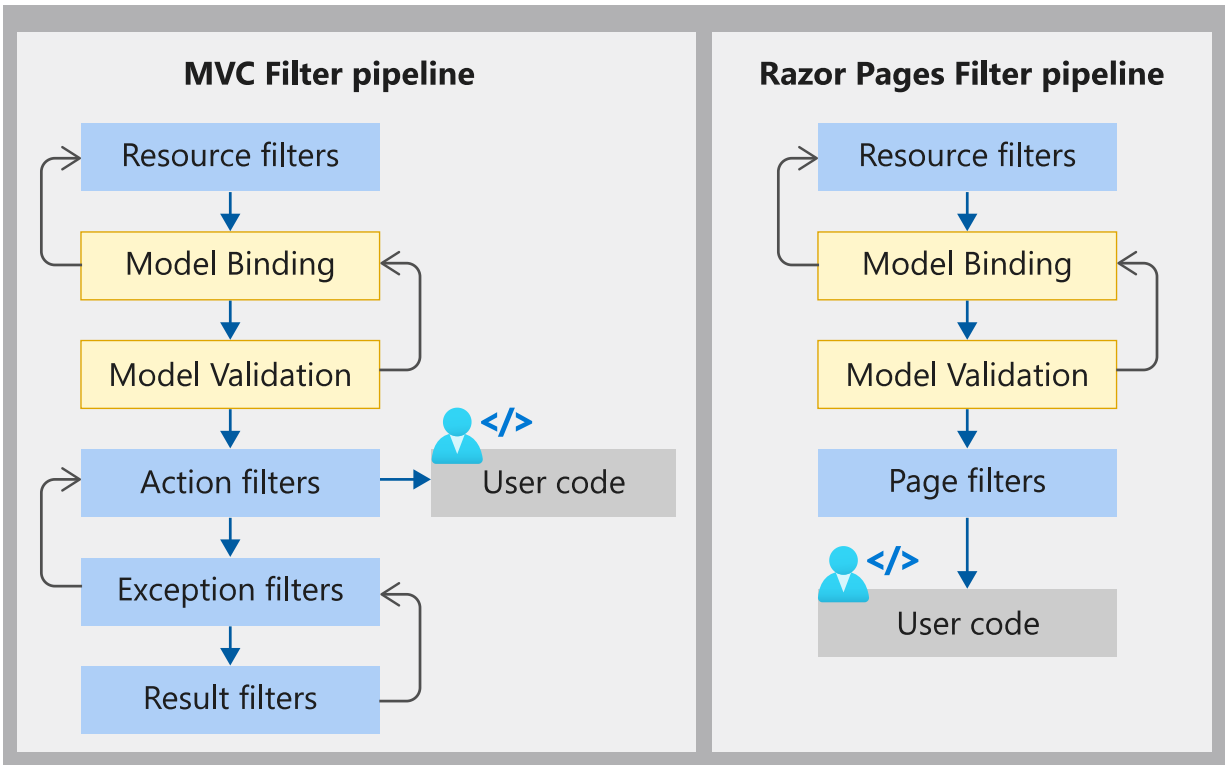
The following diagram shows the complete request processing pipeline for ASP.NET Core MVC and Razor Pages apps. You can see how, in a typical app, existing middlewares are ordered and where custom middlewares are added. You have full control over how to reorder existing middlewares or inject new custom middlewares as necessary for your scenarios.



The **Endpoint** middleware in the preceding diagram executes the filter pipeline for the corresponding app type—MVC or Razor Pages.

The **Routing** middleware in the preceding diagram is shown following **Static Files**. This is the order that the project templates implement by explicitly calling app.UseRouting. If you don't call `app.UseRouting`, the **Routing** middleware runs at the beginning of the pipeline by default. For more information, see Routing.

MVC Endpoint
(called by the Endpoint Middleware)



The order that middleware components are added in the `Program.cs` file defines the order in which the middleware components are invoked on requests and the reverse order for the response. The order is **critical** for security, performance, and functionality.

The following highlighted code in `Program.cs` adds security-related middleware components in the typical recommended order:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebMiddleware.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection")
    ?? throw new InvalidOperationException("Connection string
'DefaultConnection' not found.");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();
```

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

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

app.UseHttpsRedirection();
app.UseStaticFiles();
// app.UseCookiePolicy();

app.UseRouting();
// app.UseRateLimiter();
// app.UseRequestLocalization();
// app.UseCors();

app.UseAuthentication();
app.UseAuthorization();
// app.UseSession();
// app.UseResponseCompression();
// app.UseResponseCaching();
app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

In the preceding code:

- Middleware that is not added when creating a new web app with individual users accounts is commented out.
- Not every middleware appears in this exact order, but many do. For example:
  - `UseCors`, `UseAuthentication`, and `UseAuthorization` must appear in the order shown.
  - `UseCors` currently must appear before `UseResponseCaching`. This requirement is explained in GitHub issue dotnet/aspnetcore #23218.
  - `UseRequestLocalization` must appear before any middleware that might check the request culture, for example, `app.UseStaticFiles()`.
  - UseRateLimiter must be called after `UseRouting` when rate limiting endpoint specific APIs are used. For example, if the [EnableRateLimiting] attribute is used, `UseRateLimiter` must be called after `UseRouting`. When calling only global limiters, `UseRateLimiter` can be called before `UseRouting`.

In some scenarios, middleware has different ordering. For example, caching and compression ordering is scenario specific, and there are multiple valid orderings. For example:

```
C#

app.UseResponseCaching();
app.UseResponseCompression();
```

With the preceding code, CPU usage could be reduced by caching the compressed response, but you might end up caching multiple representations of a resource using different compression algorithms such as Gzip or Brotli.

The following ordering combines static files to allow caching compressed static files:

```
C#

app.UseResponseCaching();
app.UseResponseCompression();
app.UseStaticFiles();
```

The following `Program.cs` code adds middleware components for common app scenarios:

1. Exception/error handling

    - When the app runs in the Development environment:
        - Developer Exception Page Middleware (UseDeveloperExceptionPage) reports app runtime errors.
        - Database Error Page Middleware (UseDatabaseErrorPage) reports database runtime errors.
    - When the app runs in the Production environment:
        - Exception Handler Middleware (UseExceptionHandler) catches exceptions thrown in the following middlewares.
        - HTTP Strict Transport Security Protocol (HSTS) Middleware (UseHsts) adds the `Strict-Transport-Security` header.

2. HTTPS Redirection Middleware (UseHttpsRedirection) redirects HTTP requests to HTTPS.
3. Static File Middleware (UseStaticFiles) returns static files and short-circuits further request processing.
4. Cookie Policy Middleware (UseCookiePolicy) conforms the app to the EU General Data Protection Regulation (GDPR) regulations.

5. Routing Middleware (UseRouting) to route requests.
6. Authentication Middleware (UseAuthentication) attempts to authenticate the user before they're allowed access to secure resources.
7. Authorization Middleware (UseAuthorization) authorizes a user to access secure resources.
8. Session Middleware (UseSession) establishes and maintains session state. If the app uses session state, call Session Middleware after Cookie Policy Middleware and before MVC Middleware.
9. Endpoint Routing Middleware (UseEndpoints with MapRazorPages) to add Razor Pages endpoints to the request pipeline.

C#

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseDatabaseErrorPage();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseCookiePolicy();
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.UseSession();
app.MapRazorPages();
```

In the preceding example code, each middleware extension method is exposed on WebApplicationBuilder through the Microsoft.AspNetCore.Builder namespace.

UseExceptionHandler is the first middleware component added to the pipeline. Therefore, the Exception Handler Middleware catches any exceptions that occur in later calls.

Static File Middleware is called early in the pipeline so that it can handle requests and short-circuit without going through the remaining components. The Static File Middleware provides **no** authorization checks. Any files served by Static File Middleware, including those under *wwwroot*, are publicly available. For an approach to secure static files, see Static files in ASP.NET Core.

If the request isn't handled by the Static File Middleware, it's passed on to the Authentication Middleware (UseAuthentication), which performs authentication. Authentication doesn't short-circuit unauthenticated requests. Although Authentication Middleware authenticates requests, authorization (and rejection) occurs only after MVC selects a specific Razor Page or MVC controller and action.

The following example demonstrates a middleware order where requests for static files are handled by Static File Middleware before Response Compression Middleware. Static files aren't compressed with this middleware order. The Razor Pages responses can be compressed.

C#

```
// Static files aren't compressed by Static File Middleware.
app.UseStaticFiles();

app.UseRouting();

app.UseResponseCompression();

app.MapRazorPages();
```

For information about Single Page Applications, see Overview of Single Page Apps (SPAs) in ASP.NET Core.

UseCors and UseStaticFiles order

The order for calling `UseCors` and `UseStaticFiles` depends on the app. For more information, see UseCors and UseStaticFiles order

Forwarded Headers Middleware order

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. To run Forwarded Headers Middleware after diagnostics and error handling middleware, see Forwarded Headers Middleware order.

Branch the middleware pipeline

Map extensions are used as a convention for branching the pipeline. `Map` branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

C#

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

app.Map("/map1", HandleMapTest1);

app.Map("/map2", HandleMapTest2);

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from non-Map delegate.");
});

app.Run();

static void HandleMapTest1(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Map Test 1");
    });
}

static void HandleMapTest2(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Map Test 2");
    });
}
```

The following table shows the requests and responses from `http://localhost:1234` using the preceding code.

Expand table

Request	Response
localhost:1234	Hello from non-Map delegate.
localhost:1234/map1	Map Test 1
localhost:1234/map2	Map Test 2
localhost:1234/map3	Hello from non-Map delegate.

When `Map` is used, the matched path segments are removed from `HttpRequest.Path` and appended to `HttpRequest.PathBase` for each request.

`Map` supports nesting, for example:

C#

```
app.Map("/level1", level1App => {
    level1App.Map("/level2a", level2AApp => {
        // "/level1/level2a" processing
    });
    level1App.Map("/level2b", level2BApp => {
        // "/level1/level2b" processing
    });
});
```

`Map` can also match multiple segments at once:

C#

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

app.Map("/map1/seg1", HandleMultiSeg);

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from non-Map delegate.");
});

app.Run();

static void HandleMultiSeg(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Map Test 1");
    });
}
```

MapWhen branches the request pipeline based on the result of the given predicate. Any predicate of type `Func<HttpContext, bool>` can be used to map requests to a new branch of the pipeline. In the following example, a predicate is used to detect the presence of a query string variable `branch`:

C#

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

app.MapWhen(context => context.Request.Query.ContainsKey("branch"),
HandleBranch);
```

```
app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from non-Map delegate.");
});

app.Run();

static void HandleBranch(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        var branchVer = context.Request.Query["branch"];
        await context.Response.WriteAsync($"Branch used = {branchVer}");
    });
}
```

The following table shows the requests and responses from `http://localhost:1234` using the previous code:

Expand table

Request	Response
`localhost:1234`	`Hello from non-Map delegate.`
`localhost:1234/?branch=main`	`Branch used = main`

UseWhen also branches the request pipeline based on the result of the given predicate. Unlike with `MapWhen`, this branch is rejoined to the main pipeline if it doesn't short-circuit or contain a terminal middleware:

C#

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

app.UseWhen(context => context.Request.Query.ContainsKey("branch"),
    appBuilder => HandleBranchAndRejoin(appBuilder));

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from non-Map delegate.");
});

app.Run();

void HandleBranchAndRejoin(IApplicationBuilder app)
{
    var logger =
```

```
app.ApplicationServices.GetRequiredService<ILogger<Program>>();

    app.Use(async (context, next) =>
    {
        var branchVer = context.Request.Query["branch"];
        logger.LogInformation("Branch used = {branchVer}", branchVer);

        // Do work that doesn't write to the Response.
        await next();
        // Do other work that doesn't write to the Response.
    });
}
```

In the preceding example, a response of `Hello from non-Map delegate.` is written for all requests. If the request includes a query string variable `branch`, its value is logged before the main pipeline is rejoined.

Built-in middleware

ASP.NET Core ships with the following middleware components. The *Order* column provides notes on middleware placement in the request processing pipeline and under what conditions the middleware may terminate request processing. When a middleware short-circuits the request processing pipeline and prevents further downstream middleware from processing a request, it's called a *terminal middleware*. For more information on short-circuiting, see the Create a middleware pipeline with WebApplication section.

Expand table

Middleware	Description	Order
Authentication	Provides authentication support.	Before `HttpContext.User` is needed. Terminal for OAuth callbacks.
Authorization	Provides authorization support.	Immediately after the Authentication Middleware.
Cookie Policy	Tracks consent from users for storing personal information and enforces minimum standards for cookie fields, such	Before middleware that issues cookies. Examples: Authentication, Session, MVC (TempData).

Middleware	Description	Order
	as `secure` and `SameSite`.	
CORS	Configures Cross-Origin Resource Sharing.	Before components that use CORS. `UseCors` currently must go before `UseResponseCaching` due to this bug.
DeveloperExceptionPage	Generates a page with error information that is intended for use only in the Development environment.	Before components that generate errors. The project templates automatically register this middleware as the first middleware in the pipeline when the environment is Development.
Diagnostics	Several separate middlewares that provide a developer exception page, exception handling, status code pages, and the default web page for new apps.	Before components that generate errors. Terminal for exceptions or serving the default web page for new apps.
Forwarded Headers	Forwards proxied headers onto the current request.	Before components that consume the updated fields. Examples: scheme, host, client IP, method.
Health Check	Checks the health of an ASP.NET Core app and its dependencies, such as checking database availability.	Terminal if a request matches a health check endpoint.
Header Propagation	Propagates HTTP headers from the incoming request to the outgoing HTTP Client requests.	
HTTP Logging	Logs HTTP Requests and Responses.	At the beginning of the middleware pipeline.
HTTP Method Override	Allows an incoming POST request to override the method.	Before components that consume the updated method.
HTTPS Redirection	Redirect all HTTP requests to HTTPS.	Before components that consume the URL.

Middleware	Description	Order
HTTP Strict Transport Security (HSTS)	Security enhancement middleware that adds a special response header.	Before responses are sent and after components that modify requests. Examples: Forwarded Headers, URL Rewriting.
MVC	Processes requests with MVC/Razor Pages.	Terminal if a request matches a route.
OWIN	Interop with OWIN-based apps, servers, and middleware.	Terminal if the OWIN Middleware fully processes the request.
Output Caching	Provides support for caching responses based on configuration.	Before components that require caching. `UseRouting` must come before `UseOutputCaching`. `UseCORS` must come before `UseOutputCaching`.
Response Caching	Provides support for caching responses. This requires client participation to work. Use output caching for complete server control.	Before components that require caching. `UseCORS` must come before `UseResponseCaching`. Is typically not beneficial for UI apps such as Razor Pages because browsers generally set request headers that prevent caching. Output caching benefits UI apps.
Request Decompression	Provides support for decompressing requests.	Before components that read the request body.
Response Compression	Provides support for compressing responses.	Before components that require compression.
Request Localization	Provides localization support.	Before localization sensitive components. Must appear after Routing Middleware when using RouteDataRequestCultureProvider.
Request Timeouts	Provides support for configuring request timeouts, global and per endpoint.	`UseRequestTimeouts` must come after `UseExceptionHandler`, `UseDeveloperExceptionPage`, and `UseRouting`.
Endpoint Routing	Defines and constrains request routes.	Terminal for matching routes.

Middleware	Description	Order
SPA	Handles all requests from this point in the middleware chain by returning the default page for the Single Page Application (SPA)	Late in the chain, so that other middleware for serving static files, MVC actions, etc., takes precedence.
Session	Provides support for managing user sessions.	Before components that require Session.
Static Files	Provides support for serving static files and directory browsing.	Terminal if a request matches a file.
URL Rewrite	Provides support for rewriting URLs and redirecting requests.	Before components that consume the URL.
W3CLogging	Generates server access logs in the W3C Extended Log File Format.	At the beginning of the middleware pipeline.
WebSockets	Enables the WebSockets protocol.	Before components that are required to accept WebSocket requests.

Additional resources

- Lifetime and registration options contains a complete sample of middleware with *scoped*, *transient*, and *singleton* lifetime services.
- Write custom ASP.NET Core middleware
- Test ASP.NET Core middleware
- Configure gRPC-Web in ASP.NET Core
- Migrate HTTP handlers and modules to ASP.NET Core middleware
- App startup in ASP.NET Core
- Request Features in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Middleware activation with a third-party container in ASP.NET Core

Rate limiting middleware in ASP.NET Core

Article • 06/17/2024

By Arvin Kahbazi, Maarten Balliauw, and Rick Anderson

The `Microsoft.AspNetCore.RateLimiting` middleware provides rate limiting middleware. Apps configure rate limiting policies and then attach the policies to endpoints. Apps using rate limiting should be carefully load tested and reviewed before deploying. See Testing endpoints with rate limiting in this article for more information.

For an introduction to rate limiting, see Rate limiting middleware.

Rate limiter algorithms

The RateLimiterOptionsExtensions class provides the following extension methods for rate limiting:

- Fixed window
- Sliding window
- Token bucket
- Concurrency

Fixed window limiter

The AddFixedWindowLimiter method uses a fixed time window to limit requests. When the time window expires, a new time window starts and the request limit is reset.

Consider the following code:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: "fixed", options =>
    {
        options.PermitLimit = 4;
        options.Window = TimeSpan.FromSeconds(12);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
```

```
        options.QueueLimit = 2;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Hello {GetTicks()}"))
                          .RequireRateLimiting("fixed");

app.Run();
```

The preceding code:

- Calls AddRateLimiter to add a rate limiting service to the service collection.
- Calls `AddFixedWindowLimiter` to create a fixed window limiter with a policy name of `"fixed"` and sets:
- PermitLimit to 4 and the time Window to 12. A maximum of 4 requests per each 12-second window are allowed.
- QueueProcessingOrder to OldestFirst.
- QueueLimit to 2.
- Calls UseRateLimiter to enable rate limiting.

Apps should use Configuration to set limiter options. The following code updates the preceding code using MyRateLimitOptions for configuration:

C#

```
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);
builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
```

```
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Fixed Window Limiter {GetTicks()}"))
                            .RequireRateLimiting(fixedPolicy);

app.Run();
```

UseRateLimiter must be called after `UseRouting` when rate limiting endpoint specific APIs are used. For example, if the [EnableRateLimiting] attribute is used, `UseRateLimiter` must be called after `UseRouting`. When calling only global limiters, `UseRateLimiter` can be called before `UseRouting`.

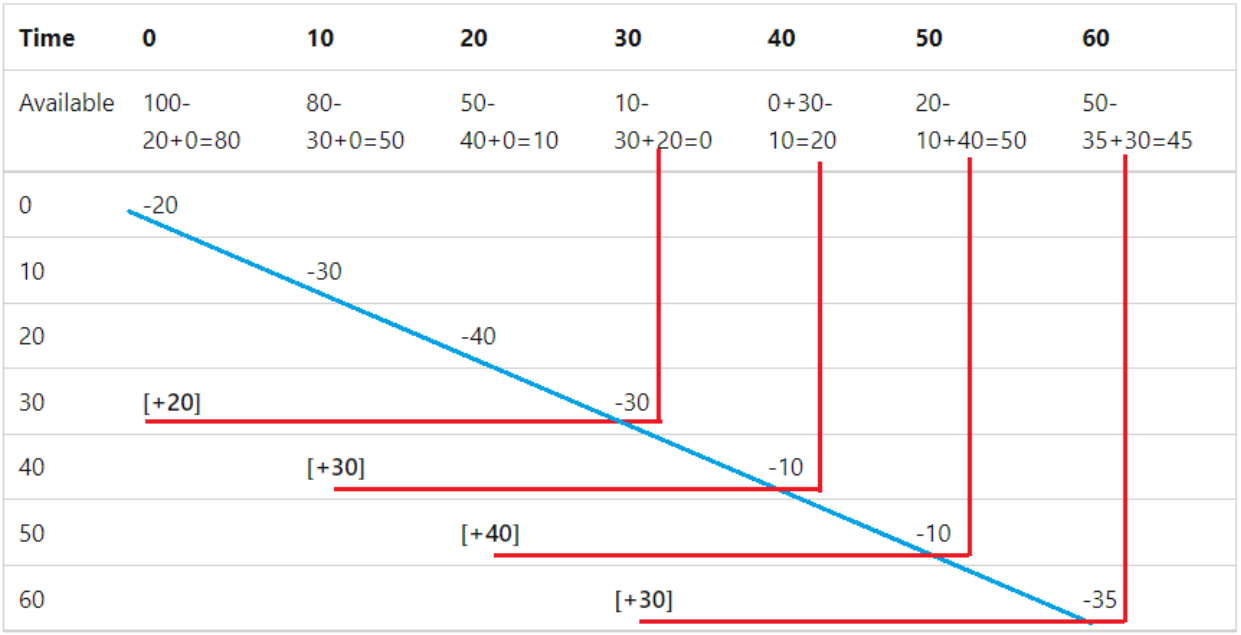
Sliding window limiter

A sliding window algorithm:

- Is similar to the fixed window limiter but adds segments per window. The window slides one segment each segment interval. The segment interval is (window time)/(segments per window).
- Limits the requests for a window to `permitLimit` requests.
- Each time window is divided in `n` segments per window.
- Requests taken from the expired time segment one window back (`n` segments prior to the current segment) are added to the current segment. We refer to the most expired time segment one window back as the expired segment.

Consider the following table that shows a sliding window limiter with a 30-second window, three segments per window, and a limit of 100 requests:

- The top row and first column shows the time segment.
- The second row shows the remaining requests available. The remaining requests are calculated as the available requests minus the processed requests plus the recycled requests.
- Requests at each time moves along the diagonal blue line.
- From time 30 on, the request taken from the expired time segment are added back to the request limit, as shown in the red lines.



The following table shows the data in the previous graph in a different format. The **Available** column shows the requests available from the previous segment (The **Carry over** from the previous row). The first row shows 100 available requests because there's no previous segment.

Expand table

Time	Available	Taken	Recycled from expired	Carry over
0	100	20	0	80
10	80	30	0	50
20	50	40	0	10
30	10	30	20	0
40	0	10	30	20
50	20	10	40	50
60	50	35	30	45

The following code uses the sliding window rate limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

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

```
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Sliding Window Limiter {GetTicks()}"))
                          .RequireRateLimiting(slidingPolicy);

app.Run();
```

Token bucket limiter

The token bucket limiter is similar to the sliding window limiter, but rather than adding back the requests taken from the expired segment, a fixed number of tokens are added each replenishment period. The tokens added each segment can't increase the available tokens to a number higher than the token bucket limit. The following table shows a token bucket limiter with a limit of 100 tokens and a 10-second replenishment period.

Expand table

Time	Available	Taken	Added	Carry over
0	100	20	0	80
10	80	10	20	90
20	90	5	15	100
30	100	30	20	90
40	90	6	16	100
50	100	40	20	80

Time	Available	Taken	Added	Carry over
60	80	50	20	50

The following code uses the token bucket limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

var tokenPolicy = "token";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(_ => _
    .AddTokenBucketLimiter(policyName: tokenPolicy, options =>
    {
        options.TokenLimit = myOptions.TokenLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
        options.ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod);
        options.TokensPerPeriod = myOptions.TokensPerPeriod;
        options.AutoReplenishment = myOptions.AutoReplenishment;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Token Limiter {GetTicks()}"))
                            .RequireRateLimiting(tokenPolicy);

app.Run();
```

When AutoReplenishment is set to `true`, an internal timer replenishes the tokens every ReplenishmentPeriod; when set to `false`, the app must call TryReplenish on the limiter.

Concurrency limiter

The concurrency limiter limits the number of concurrent requests. Each request reduces the concurrency limit by one. When a request completes, the limit is increased by one.

Unlike the other requests limiters that limit the total number of requests for a specified period, the concurrency limiter limits only the number of concurrent requests and doesn't cap the number of requests in a time period.

The following code uses the concurrency limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

var concurrencyPolicy = "Concurrency";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(_ => _
    .AddConcurrencyLimiter(policyName: concurrencyPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", async () =>
{
    await Task.Delay(500);
    return Results.Ok($"Concurrency Limiter {GetTicks()}");

}).RequireRateLimiting(concurrencyPolicy);

app.Run();
```

Create chained limiters

The CreateChained API allows passing in multiple PartitionedRateLimiter which are combined into one `PartitionedRateLimiter`. The combined limiter runs all the input limiters in sequence.

The following code uses `CreateChained`:

C#

```
using System.Globalization;
using System.Threading.RateLimiting;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRateLimiter(_ =>
{
    _.OnRejected = (context, _) =>
    {
        if (context.Lease.TryGetMetadata(MetadataName.RetryAfter, out var
retryAfter))
        {
            context.HttpContext.Response.Headers.RetryAfter =
                ((int)
retryAfter.TotalSeconds).ToString(NumberFormatInfo.InvariantInfo);
        }

        context.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
        context.HttpContext.Response.WriteAsync("Too many requests. Please
try again later.");

        return new ValueTask();
    };
    _.GlobalLimiter = PartitionedRateLimiter.CreateChained(
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
        {
            var userAgent =
httpContext.Request.Headers.UserAgent.ToString();

            return RateLimitPartition.GetFixedWindowLimiter
            (userAgent, _ =>
                new FixedWindowRateLimiterOptions
                {
                    AutoReplenishment = true,
                    PermitLimit = 4,
                    Window = TimeSpan.FromSeconds(2)
                });
        }),
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
        {
            var userAgent =
httpContext.Request.Headers.UserAgent.ToString();

            return RateLimitPartition.GetFixedWindowLimiter
            (userAgent, _ =>
                new FixedWindowRateLimiterOptions
                {
                    AutoReplenishment = true,
                    PermitLimit = 20,
                    Window = TimeSpan.FromSeconds(30)
                });
```

```
        }));
});

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

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Hello {GetTicks()}"));

app.Run();
```

For more information, see the CreateChained source code

`EnableRateLimiting` and `DisableRateLimiting` attributes

The [EnableRateLimiting] and [DisableRateLimiting] attributes can be applied to a Controller, action method, or Razor Page. For Razor Pages, the attribute must be applied to the Razor Page and not the page handlers. For example, `[EnableRateLimiting]` can't be applied to `OnGet`, `OnPost`, or any other page handler.

The `[DisableRateLimiting]` attribute ***disables*** rate limiting to the Controller, action method, or Razor Page regardless of named rate limiters or global limiters applied. For example, consider the following code which calls RequireRateLimiting to apply the `fixedPolicy` rate limiting to all controller endpoints:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
```

```
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.SlidingPermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

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

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

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

app.MapRazorPages().RequireRateLimiting(slidingPolicy);
app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy);

app.Run();
```

In the following code, `[DisableRateLimiting]` disables rate limiting and overrides `[EnableRateLimiting("fixed")]` applied to the `Home2Controller` and `app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy)` called in `Program.cs`:

C#

```
[EnableRateLimiting("fixed")]
public class Home2Controller : Controller
{
    private readonly ILogger<Home2Controller> _logger;

    public Home2Controller(ILogger<Home2Controller> logger)
    {
        _logger = logger;
```

```
    }

    public ActionResult Index()
    {
        return View();
    }

    [EnableRateLimiting("sliding")]
    public ActionResult Privacy()
    {
        return View();
    }

    [DisableRateLimiting]
    public ActionResult NoLimit()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
HttpContext.TraceIdentifier });
    }
}
```

In the preceding code, the `[EnableRateLimiting("sliding")]` is ***not*** applied to the `Privacy` action method because `Program.cs` called `app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy)`.

Consider the following code which doesn't call `RequireRateLimiting` on `MapRazorPages` or `MapDefaultControllerRoute`:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
```

```
ons);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.SlidingPermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

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

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

app.MapRazorPages();
app.MapDefaultControllerRoute();  // RequireRateLimiting not called

app.Run();
```

Consider the following controller:

C#

```
[EnableRateLimiting("fixed")]
public class Home2Controller : Controller
{
    private readonly ILogger<Home2Controller> _logger;

    public Home2Controller(ILogger<Home2Controller> logger)
    {
```

```
        _logger = logger;
    }

    public ActionResult Index()
    {
        return View();
    }

    [EnableRateLimiting("sliding")]
    public ActionResult Privacy()
    {
        return View();
    }

    [DisableRateLimiting]
    public ActionResult NoLimit()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
HttpContext.TraceIdentifier });
    }
}
```

In the preceding controller:

- The `"fixed"` policy rate limiter is applied to all action methods that don't have `EnableRateLimiting` and `DisableRateLimiting` attributes.
- The `"sliding"` policy rate limiter is applied to the `Privacy` action.
- Rate limiting is disabled on the `NoLimit` action method.

Applying attributes to Razor Pages

For Razor Pages, the attribute must be applied to the Razor Page and not the page handlers. For example, `[EnableRateLimiting]` can't be applied to `OnGet`, `OnPost`, or any other page handler.

The `DisableRateLimiting` attribute disables rate limiting on a Razor Page. `EnableRateLimiting` is only applied to a Razor Page if `MapRazorPages().RequireRateLimiting(Policy)` has ***not*** been called.

Limiter algorithm comparison

The fixed, sliding, and token limiters all limit the maximum number of requests in a time period. The concurrency limiter limits only the number of concurrent requests and doesn't cap the number of requests in a time period. The cost of an endpoint should be considered when selecting a limiter. The cost of an endpoint includes the resources used, for example, time, data access, CPU, and I/O.

Rate limiter samples

The following samples aren't meant for production code but are examples on how to use the limiters.

Limiter with `OnRejected`, `RetryAfter`, and `GlobalLimiter`

The following sample:

- Creates a RateLimiterOptions.OnRejected callback that is called when a request exceeds the specified limit. `retryAfter` can be used with the TokenBucketRateLimiter, FixedWindowLimiter, and SlidingWindowLimiter because these algorithms are able to estimate when more permits will be added. The `ConcurrencyLimiter` has no way of calculating when permits will be available.
- Adds the following limiters:
  - A `SampleRateLimiterPolicy` which implements the `IRateLimiterPolicy<TPartitionKey>` interface. The `SampleRateLimiterPolicy` class is shown later in this article.
  - A `SlidingWindowLimiter`:
    - With a partition for each authenticated user.
    - One shared partition for all anonymous users.
  - A GlobalLimiter that is applied to all requests. The global limiter will be executed first, followed by the endpoint-specific limiter, if one exists. The `GlobalLimiter` creates a partition for each IPAddress.

C#

```
// Preceding code removed for brevity.
using System.Globalization;
using System.Net;
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebRateLimitAuth;
using WebRateLimitAuth.Data;
using WebRateLimitAuth.Models;
```

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

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection") ??
    throw new InvalidOperationException("Connection string
'DefaultConnection' not found.");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var userPolicyName = "user";
var helloPolicy = "hello";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(limiterOptions =>
{
    limiterOptions.OnRejected = (context, cancellationToken) =>
    {
        if (context.Lease.TryGetMetadata(MetadataName.RetryAfter, out var
retryAfter))
        {
            context.HttpContext.Response.Headers.RetryAfter =
                ((int)
retryAfter.TotalSeconds).ToString(NumberFormatInfo.InvariantInfo);
        }

        context.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
        context.HttpContext.RequestServices.GetService<ILoggerFactory>()?
            .CreateLogger("Microsoft.AspNetCore.RateLimitingMiddleware")
            .LogWarning("OnRejected: {GetUserEndPoint}",
GetUserEndPoint(context.HttpContext));

        return new ValueTask();
    };

    limiterOptions.AddPolicy<string, SampleRateLimiterPolicy>(helloPolicy);
    limiterOptions.AddPolicy(userPolicyName, context =>
    {
        var username = "anonymous user";
        if (context.User.Identity?.IsAuthenticated is true)
        {
```

```
            username = context.User.ToString()!;
        }

        return RateLimitPartition.GetSlidingWindowLimiter(username,
            _ => new SlidingWindowRateLimiterOptions
            {
                PermitLimit = myOptions.PermitLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                Window = TimeSpan.FromSeconds(myOptions.Window),
                SegmentsPerWindow = myOptions.SegmentsPerWindow
            });

    });

    limiterOptions.GlobalLimiter =
PartitionedRateLimiter.Create<HttpContext, IPAddress>(context =>
    {
        IPAddress? remoteIpAddress = context.Connection.RemoteIpAddress;

        if (!IPAddress.IsLoopback(remoteIpAddress!))
        {
            return RateLimitPartition.GetTokenBucketLimiter
            (remoteIpAddress!, _ =>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetNoLimiter(IPAddress.Loopback);
    });
});

var app = builder.Build();

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

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

```
app.UseRouting();
app.UseRateLimiter();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages().RequireRateLimiting(userPolicyName);
app.MapDefaultControllerRoute();

static string GetUserEndPoint(HttpContext context) =>
   $"User {context.User.Identity?.Name ?? "Anonymous"} endpoint:
{context.Request.Path}"
   + $" {context.Connection.RemoteIpAddress}";
static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/a", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}")
    .RequireRateLimiting(userPolicyName);

app.MapGet("/b", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}")
    .RequireRateLimiting(helloPolicy);

app.MapGet("/c", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}");

app.Run();
```

> ⚠ **Warning**
>
> Creating partitions on client IP addresses makes the app vulnerable to Denial of Service Attacks which employ IP Source Address Spoofing. For more information, see BCP 38 RFC 2827 Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing.

See the samples repository for the complete Program.cs file.

The `SampleRateLimiterPolicy` class

C#

```
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.RateLimiting;
using Microsoft.Extensions.Options;
using WebRateLimitAuth.Models;

namespace WebRateLimitAuth;

public class SampleRateLimiterPolicy : IRateLimiterPolicy<string>
```

```
{
    private Func<OnRejectedContext, CancellationToken, ValueTask>?
_onRejected;
    private readonly MyRateLimitOptions _options;

    public SampleRateLimiterPolicy(ILogger<SampleRateLimiterPolicy> logger,
                                   IOptions<MyRateLimitOptions> options)
    {
        _onRejected = (ctx, token) =>
        {
            ctx.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
            logger.LogWarning($"Request rejected by
{nameof(SampleRateLimiterPolicy)}");
            return ValueTask.CompletedTask;
        };
        _options = options.Value;
    }

    public Func<OnRejectedContext, CancellationToken, ValueTask>? OnRejected
=> _onRejected;

    public RateLimitPartition<string> GetPartition(HttpContext httpContext)
    {
        return RateLimitPartition.GetSlidingWindowLimiter(string.Empty,
            _ => new SlidingWindowRateLimiterOptions
            {
                PermitLimit = _options.PermitLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = _options.QueueLimit,
                Window = TimeSpan.FromSeconds(_options.Window),
                SegmentsPerWindow = _options.SegmentsPerWindow
            });
    }
}
```

In the preceding code, OnRejected uses OnRejectedContext to set the response status to 429 Too Many Requests. The default rejected status is 503 Service Unavailable.

Limiter with authorization

The following sample uses JSON Web Tokens (JWT) and creates a partition with the JWT access token. In a production app, the JWT would typically be provided by a server acting as a Security token service (STS). For local development, the dotnet user-jwts command line tool can be used to create and manage app-specific local JWTs.

C#

```
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.Authentication;
```

```
using Microsoft.Extensions.Primitives;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var jwtPolicyName = "jwt";

builder.Services.AddRateLimiter(limiterOptions =>
{
    limiterOptions.RejectionStatusCode =
StatusCodes.Status429TooManyRequests;
    limiterOptions.AddPolicy(policyName: jwtPolicyName, partitioner:
httpContext =>
    {
        var accessToken =
httpContext.Features.Get<IAuthenticateResultFeature>()?

.AuthenticateResult?.Properties?.GetTokenValue("access_token")?.ToString()
                           ?? string.Empty;

        if (!StringValues.IsNullOrEmpty(accessToken))
        {
            return RateLimitPartition.GetTokenBucketLimiter(accessToken, _
=>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetTokenBucketLimiter("Anon", _ =>
            new TokenBucketRateLimiterOptions
            {
                TokenLimit = myOptions.TokenLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                TokensPerPeriod = myOptions.TokensPerPeriod,
                AutoReplenishment = true
            });
    });
});
```

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

app.UseAuthorization();
app.UseRateLimiter();

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

app.MapGet("/jwt", (HttpContext context) => $"Hello
{GetUserEndPointMethod(context)}")
    .RequireRateLimiting(jwtPolicyName)
    .RequireAuthorization();

app.MapPost("/post", (HttpContext context) => $"Hello
{GetUserEndPointMethod(context)}")
    .RequireRateLimiting(jwtPolicyName)
    .RequireAuthorization();

app.Run();

static string GetUserEndPointMethod(HttpContext context) =>
    $"Hello {context.User.Identity?.Name ?? "Anonymous"} " +
    $"Endpoint:{context.Request.Path} Method: {context.Request.Method}";
```

Limiter with `ConcurrencyLimiter`, `TokenBucketRateLimiter`, and authorization

The following sample:

- Adds a `ConcurrencyLimiter` with a policy name of `"get"` that is used on the Razor Pages.
- Adds a `TokenBucketRateLimiter` with a partition for each authorized user and a partition for all anonymous users.
- Sets RateLimiterOptions.RejectionStatusCode to 429 Too Many Requests.

C#

```
var getPolicyName = "get";
var postPolicyName = "post";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(_ => _
    .AddConcurrencyLimiter(policyName: getPolicyName, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
```

```
    })
    .AddPolicy(policyName: postPolicyName, partitioner: httpContext =>
    {
        string userName = httpContext.User.Identity?.Name ?? string.Empty;

        if (!StringValues.IsNullOrEmpty(userName))
        {
            return RateLimitPartition.GetTokenBucketLimiter(userName, _ =>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetTokenBucketLimiter("Anon", _ =>
            new TokenBucketRateLimiterOptions
            {
                TokenLimit = myOptions.TokenLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                TokensPerPeriod = myOptions.TokensPerPeriod,
                AutoReplenishment = true
            });
    }));
```

See the samples repository for the complete Program.cs file.

Testing endpoints with rate limiting

Before deploying an app using rate limiting to production, stress test the app to validate the rate limiters and options used. For example, create a JMeter script with a tool like BlazeMeter or Apache JMeter HTTP(S) Test Script Recorder and load the script to Azure Load Testing.

Creating partitions with user input makes the app vulnerable to Denial of Service (DoS) Attacks. For example, creating partitions on client IP addresses makes the app vulnerable to Denial of Service Attacks that employ IP Source Address Spoofing. For more information, see BCP 38 RFC 2827 Network Ingress Filtering: Defeating Denial of Service Attacks that employ IP Source Address Spoofing.

Additional resources

- Rate limiting middleware by Maarten Balliauw provides an excellent introduction and overview to rate limiting.
- Rate limit an HTTP handler in .NET

Middleware in Minimal API apps

Article • 07/26/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

WebApplication automatically adds the following middleware in Minimal API applications depending on certain conditions:

- UseDeveloperExceptionPage is added first when the HostingEnvironment is `"Development"`.
- UseRouting is added second if user code didn't already call `UseRouting` and if there are endpoints configured, for example `app.MapGet`.
- UseEndpoints is added at the end of the middleware pipeline if any endpoints are configured.
- UseAuthentication is added immediately after `UseRouting` if user code didn't already call `UseAuthentication` and if IAuthenticationSchemeProvider can be detected in the service provider. `IAuthenticationSchemeProvider` is added by default when using AddAuthentication, and services are detected using IServiceProviderIsService.
- UseAuthorization is added next if user code didn't already call `UseAuthorization` and if IAuthorizationHandlerProvider can be detected in the service provider. `IAuthorizationHandlerProvider` is added by default when using AddAuthorization, and services are detected using `IServiceProviderIsService`.
- User configured middleware and endpoints are added between `UseRouting` and `UseEndpoints`.

The following code is effectively what the automatic middleware being added to the app produces:

C#

```
if (isDevelopment)
{
    app.UseDeveloperExceptionPage();
}
```

```
app.UseRouting();

if (isAuthenticationConfigured)
{
    app.UseAuthentication();
}

if (isAuthorizationConfigured)
{
    app.UseAuthorization();
}

// user middleware/endpoints
app.CustomMiddleware(...);
app.MapGet("/", () => "hello world");
// end user middleware/endpoints

app.UseEndpoints(e => {});
```

In some cases, the default middleware configuration isn't correct for the app and requires modification. For example, UseCors should be called before UseAuthentication and UseAuthorization. The app needs to call `UseAuthentication` and `UseAuthorization` if `UseCors` is called:

C#

```
app.UseCors();
app.UseAuthentication();
app.UseAuthorization();
```

If middleware should be run before route matching occurs, UseRouting should be called and the middleware should be placed before the call to `UseRouting`. UseEndpoints isn't required in this case as it is automatically added as described previously:

C#

```
app.Use((context, next) =>
{
    return next(context);
});

app.UseRouting();

// other middleware and endpoints
```

When adding a terminal middleware:

- The middleware must be added after `UseEndpoints`.
- The app needs to call `UseRouting` and `UseEndpoints` so that the terminal middleware can be placed at the correct location.

C#

```
app.UseRouting();

app.MapGet("/", () => "hello world");

app.UseEndpoints(e => {});

app.Run(context =>
{
    context.Response.StatusCode = 404;
    return Task.CompletedTask;
});
```

Terminal middleware is middleware that runs if no endpoint handles the request.

For information on antiforgery middleware in Minimal APIs, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core

For more information about middleware see ASP.NET Core Middleware, and the list of built-in middleware that can be added to applications.

For more information about Minimal APIs see Minimal APIs overview.

Test ASP.NET Core middleware

Article • 01/11/2024

By Chris Ross

Middleware can be tested in isolation with TestServer. It allows you to:

- Instantiate an app pipeline containing only the components that you need to test.
- Send custom requests to verify middleware behavior.

Advantages:

- Requests are sent in-memory rather than being serialized over the network.
- This avoids additional concerns, such as port management and HTTPS certificates.
- Exceptions in the middleware can flow directly back to the calling test.
- It's possible to customize server data structures, such as HttpContext, directly in the test.

Set up the TestServer

In the test project, create a test:

- Build and start a host that uses TestServer.
- Add any required services that the middleware uses.
- Add a package reference to the project for the Microsoft.AspNetCore.TestHost NuGet package.
- Configure the processing pipeline to use the middleware for the test.

```
C#

[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
```

```
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    ...
}
```

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Send requests with HttpClient

Send a request using HttpClient:

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();
```

```
    var response = await host.GetTestClient().GetAsync("/");

    ...
}
```

Assert the result. First, make an assertion the opposite of the result that you expect. An initial run with a false positive assertion confirms that the test fails when the middleware is performing correctly. Run the test and confirm that the test fails.

In the following example, the middleware should return a 404 status code (*Not Found*) when the root endpoint is requested. Make the first test run with `Assert.NotEqual( ... );`, which should fail:

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    Assert.NotEqual(HttpStatusCode.NotFound, response.StatusCode);
}
```

Change the assertion to test the middleware under normal operating conditions. The final test uses `Assert.Equal( ... );`. Run the test again to confirm that it passes.

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
```

```
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}
```

Send requests with HttpContext

A test app can also send a request using SendAsync(Action<HttpContext>, CancellationToken). In the following example, several checks are made when `https://example.com/A/Path/?and=query` is processed by the middleware:

C#

```
[Fact]
public async Task TestMiddleware_ExpectedResponse()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var server = host.GetTestServer();
    server.BaseAddress = new Uri("https://example.com/A/Path/");

    var context = await server.SendAsync(c =>
```

```
    {
        c.Request.Method = HttpMethods.Post;
        c.Request.Path = "/and/file.txt";
        c.Request.QueryString = new QueryString("?and=query");
    });

    Assert.True(context.RequestAborted.CanBeCanceled);
    Assert.Equal(HttpProtocol.Http11, context.Request.Protocol);
    Assert.Equal("POST", context.Request.Method);
    Assert.Equal("https", context.Request.Scheme);
    Assert.Equal("example.com", context.Request.Host.Value);
    Assert.Equal("/A/Path", context.Request.PathBase.Value);
    Assert.Equal("/and/file.txt", context.Request.Path.Value);
    Assert.Equal("?and=query", context.Request.QueryString.Value);
    Assert.NotNull(context.Request.Body);
    Assert.NotNull(context.Request.Headers);
    Assert.NotNull(context.Response.Headers);
    Assert.NotNull(context.Response.Body);
    Assert.Equal(404, context.Response.StatusCode);
    Assert.Null(context.Features.Get<IHttpResponseFeature>().ReasonPhrase);
}
```

SendAsync permits direct configuration of an HttpContext object rather than using the HttpClient abstractions. Use SendAsync to manipulate structures only available on the server, such as HttpContext.Items or HttpContext.Features.

As with the earlier example that tested for a *404 - Not Found* response, check the opposite for each `Assert` statement in the preceding test. The check confirms that the test fails correctly when the middleware is operating normally. After you've confirmed that the false positive test works, set the final `Assert` statements for the expected conditions and values of the test. Run it again to confirm that the test passes.

Add request routes

Additional routes can be added by configuration using the test `HttpClient`:

C#

```
    [Fact]
    public async Task TestWithEndpoint_ExpectedResponse ()
    {
        using var host = await new HostBuilder()
            .ConfigureWebHost(webBuilder =>
            {
                webBuilder
                    .UseTestServer()
                    .ConfigureServices(services =>
                    {
                        services.AddRouting();
```

```
                    })
                    .Configure(app =>
                    {
                        app.UseRouting();
                        app.UseMiddleware<MyMiddleware>();
                        app.UseEndpoints(endpoints =>
                        {
                            endpoints.MapGet("/hello", () =>
                                TypedResults.Text("Hello Tests"));
                        });
                    });
            })
            .StartAsync();

        var client = host.GetTestClient();

        var response = await client.GetAsync("/hello");

        Assert.True(response.IsSuccessStatusCode);
        var responseBody = await response.Content.ReadAsStringAsync();
        Assert.Equal("Hello Tests", responseBody);
```

Additional routes can also be added using the approach `server.SendAsync`.

TestServer limitations

TestServer:

- Was created to replicate server behaviors to test middleware.
- Does ***not*** try to replicate all HttpClient behaviors.
- Attempts to give the client access to as much control over the server as possible, and with as much visibility into what's happening on the server as possible. For example it may throw exceptions not normally thrown by `HttpClient` in order to directly communicate server state.
- Doesn't set some transport specific headers by default as those aren't usually relevant to middleware. For more information, see the next section.
- Ignores the `Stream` position passed through StreamContent. HttpClient sends the entire stream from the start position, even when positioning is set. For more information, see this GitHub issue.

Content-Length and Transfer-Encoding headers

TestServer does ***not*** set transport related request or response headers such as Content-Length or Transfer-Encoding. Applications should avoid depending on these headers because their usage varies by client, scenario, and protocol. If `Content-Length`

and `Transfer-Encoding` are necessary to test a specific scenario, they can be specified in the test when composing the HttpRequestMessage or HttpContext. For more information, see the following GitHub issues:

- dotnet/aspnetcore#21677
- dotnet/aspnetcore#18463
- dotnet/aspnetcore#13273

Response Caching Middleware in ASP.NET Core

Article • 07/16/2024

By John Luo and Rick Anderson

This article explains how to configure Response Caching Middleware in an ASP.NET Core app. The middleware determines when responses are cacheable, stores responses, and serves responses from cache. For an introduction to HTTP caching and the [ResponseCache] attribute, see Response Caching.

The Response caching middleware:

- Enables caching server responses based on HTTP cache headers. Implements the standard HTTP caching semantics. Caches based on HTTP cache headers like proxies do.
- Is typically not beneficial for UI apps such as Razor Pages because browsers generally set request headers that prevent caching. Output caching, which is available in ASP.NET Core 7.0 and later, benefits UI apps. With output caching, configuration decides what should be cached independently of HTTP headers.
- May be beneficial for public GET or HEAD API requests from clients where the Conditions for caching are met.

To test response caching, use Fiddler, or another tool that can explicitly set request headers. Setting headers explicitly is preferred for testing caching. For more information, see Troubleshooting.

Configuration

In `Program.cs`, add the Response Caching Middleware services AddResponseCaching to the service collection and configure the app to use the middleware with the UseResponseCaching extension method. `UseResponseCaching` adds the middleware to the request processing pipeline:

C#

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

builder.Services.AddResponseCaching();

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

```
app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();
```

> ⚠ **Warning**
>
> UseCors must be called before UseResponseCaching when using CORS middleware.

The sample app adds headers to control caching on subsequent requests:

- Cache-Control: Caches cacheable responses for up to 10 seconds.
- Vary: Configures the middleware to serve a cached response only if the Accept-Encoding header of subsequent requests matches that of the original request.

C#

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

builder.Services.AddResponseCaching();

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
    context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next();
});

app.MapGet("/", () => DateTime.Now.Millisecond);
```

```
app.Run();
```

The preceding headers are not written to the response and are overridden when a controller, action, or Razor Page:

- Has a [ResponseCache] attribute. This applies even if a property isn't set. For example, omitting the VaryByHeader property will cause the corresponding header to be removed from the response.

Response Caching Middleware only caches server responses that result in a 200 (OK) status code. Any other responses, including error pages, are ignored by the middleware.

> ⚠ **Warning**
>
> Responses containing content for authenticated clients must be marked as not cacheable to prevent the middleware from storing and serving those responses. See **Conditions for caching** for details on how the middleware determines if a response is cacheable.

The preceding code typically doesn't return a cached value to a browser. Use Fiddler or another tool that can explicitly set request headers and is preferred for testing caching. For more information, see Troubleshooting in this article.

Options

Response caching options are shown in the following table.

Expand table

Option	Description
MaximumBodySize	The largest cacheable size for the response body in bytes. The default value is `64 * 1024 * 1024` (64 MB).
SizeLimit	The size limit for the response cache middleware in bytes. The default value is `100 * 1024 * 1024` (100 MB).
UseCaseSensitivePaths	Determines if responses are cached on case-sensitive paths. The default value is `false`.

The following example configures the middleware to:

- Cache responses with a body size smaller than or equal to 1,024 bytes.

- Store the responses by case-sensitive paths. For example, `/page1` and `/Page1` are stored separately.

C#

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

builder.Services.AddResponseCaching(options =>
{
    options.MaximumBodySize = 1024;
    options.UseCaseSensitivePaths = true;
});

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
    context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next(context);
});

app.MapGet("/", () => DateTime.Now.Millisecond);

app.Run();
```

VaryByQueryKeys

When using MVC, web API controllers, or Razor Pages page models, the [ResponseCache] attribute specifies the parameters necessary for setting the appropriate headers for response caching. The only parameter of the `[ResponseCache]` attribute that strictly requires the middleware is VaryByQueryKeys, which doesn't correspond to an actual HTTP header. For more information, see Response caching in ASP.NET Core.

When not using the `[ResponseCache]` attribute, response caching can be varied with `VaryByQueryKeys`. Use the ResponseCachingFeature directly from the HttpContext.Features:

```
C#

var responseCachingFeature =
context.HttpContext.Features.Get<IResponseCachingFeature>();

if (responseCachingFeature != null)
{
    responseCachingFeature.VaryByQueryKeys = new[] { "MyKey" };
}
```

Using a single value equal to `*` in `VaryByQueryKeys` varies the cache by all request query parameters.

HTTP headers used by Response Caching Middleware

The following table provides information on HTTP headers that affect response caching.

Expand table

Header	Details
`Authorization`	The response isn't cached if the header exists.
`Cache-Control`	The middleware only considers caching responses marked with the `public` cache directive. Control caching with the following parameters: • max-age • max-stale† • min-fresh • must-revalidate • no-cache • no-store • only-if-cached • private • public • s-maxage • proxy-revalidate‡ †If no limit is specified to `max-stale`, the middleware takes no action. ‡`proxy-revalidate` has the same effect as `must-revalidate`.

Header	Details
	For more information, see RFC 9111: Request Directives.
`Pragma`	A `Pragma: no-cache` header in the request produces the same effect as `Cache-Control: no-cache`. This header is overridden by the relevant directives in the `Cache-Control` header, if present. Considered for backward compatibility with HTTP/1.0.
`Set-Cookie`	The response isn't cached if the header exists. Any middleware in the request processing pipeline that sets one or more cookies prevents the Response Caching Middleware from caching the response (for example, the cookie-based TempData provider).
`Vary`	The `Vary` header is used to vary the cached response by another header. For example, cache responses by encoding by including the `Vary: Accept-Encoding` header, which caches responses for requests with headers `Accept-Encoding: gzip` and `Accept-Encoding: text/plain` separately. A response with a header value of `*` is never stored.
`Expires`	A response deemed stale by this header isn't stored or retrieved unless overridden by other `Cache-Control` headers.
`If-None-Match`	The full response is served from cache if the value isn't `*` and the `ETag` of the response doesn't match any of the values provided. Otherwise, a 304 (Not Modified) response is served.
`If-Modified-Since`	If the `If-None-Match` header isn't present, a full response is served from cache if the cached response date is newer than the value provided. Otherwise, a *304 - Not Modified* response is served.
`Date`	When serving from cache, the `Date` header is set by the middleware if it wasn't provided on the original response.
`Content-Length`	When serving from cache, the `Content-Length` header is set by the middleware if it wasn't provided on the original response.
`Age`	The `Age` header sent in the original response is ignored. The middleware computes a new value when serving a cached response.

Caching respects request Cache-Control directives

The middleware respects the rules of RFC 9111: HTTP Caching (Section 5.2. Cache-Control). The rules require a cache to honor a valid `Cache-Control` header sent by the client. Under the specification, a client can make requests with a `no-cache` header value

and force the server to generate a new response for every request. Currently, there's no developer control over this caching behavior when using the middleware because the middleware adheres to the official caching specification.

For more control over caching behavior, explore other caching features of ASP.NET Core. See the following topics:

- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Troubleshooting

The Response Caching Middleware uses IMemoryCache, which has a limited capacity. When the capacity is exceeded, the memory cache is compacted.

If caching behavior isn't as expected, confirm that responses are cacheable and capable of being served from the cache. Examine the request's incoming headers and the response's outgoing headers. Enable logging to help with debugging.

When testing and troubleshooting caching behavior, a browser typically sets request headers that prevent caching. For example, a browser may set the `Cache-Control` header to `no-cache` or `max-age=0` when refreshing a page. Fiddler and other tools can explicitly set request headers and are preferred for testing caching.

Conditions for caching

- The request must result in a server response with a 200 (OK) status code.
- The request method must be GET or HEAD.
- Response Caching Middleware must be placed before middleware that require caching. For more information, see ASP.NET Core Middleware.
- The `Authorization` header must not be present.
- `Cache-Control` header parameters must be valid, and the response must be marked `public` and not marked `private`.
- The `Pragma: no-cache` header must not be present if the `Cache-Control` header isn't present, as the `Cache-Control` header overrides the `Pragma` header when present.
- The `Set-Cookie` header must not be present.
- `Vary` header parameters must be valid and not equal to `*`.

- The `Content-Length` header value (if set) must match the size of the response body.
- The IHttpSendFileFeature isn't used.
- The response must not be stale as specified by the `Expires` header and the `max-age` and `s-maxage` cache directives.
- Response buffering must be successful. The size of the response must be smaller than the configured or default SizeLimit. The body size of the response must be smaller than the configured or default MaximumBodySize.
- The response must be cacheable according to RFC 9111: HTTP Caching. For example, the `no-store` directive must not exist in request or response header fields. See RFC 9111: HTTP Caching (Section 3: *Storing Responses in Caches*) for details.

> **Note**
>
> The Antiforgery system for generating secure tokens to prevent Cross-Site Request Forgery (CSRF) attacks sets the `Cache-Control` and `Pragma` headers to `no-cache` so that responses aren't cached. For information on how to disable antiforgery tokens for HTML form elements, see **Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core**.

Additional resources

- View or download sample code (how to download)
- GitHub source for IResponseCachingPolicyProvider
- GitHub source for IResponseCachingPolicyProvider
- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Write custom ASP.NET Core middleware

Article • 07/26/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Fiyaz Hasan, Rick Anderson, and Steve Smith

Middleware is software that's assembled into an app pipeline to handle requests and responses. ASP.NET Core provides a rich set of built-in middleware components, but in some scenarios you might want to write a custom middleware.

This topic describes how to write *convention-based* middleware. For an approach that uses strong typing and per-request activation, see Factory-based middleware activation in ASP.NET Core.

Middleware class

Middleware is generally encapsulated in a class and exposed with an extension method. Consider the following inline middleware, which sets the culture for the current request from a query string:

C#

```
using System.Globalization;

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

app.UseHttpsRedirection();
```

```
app.Use(async (context, next) =>
{
    var cultureQuery = context.Request.Query["culture"];
    if (!string.IsNullOrWhiteSpace(cultureQuery))
    {
        var culture = new CultureInfo(cultureQuery);

        CultureInfo.CurrentCulture = culture;
        CultureInfo.CurrentUICulture = culture;
    }

    // Call the next delegate/middleware in the pipeline.
    await next(context);
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync(
        $"CurrentCulture.DisplayName:
{CultureInfo.CurrentCulture.DisplayName}");
});

app.Run();
```

The preceding highlighted inline middleware is used to demonstrate creating a middleware component by calling Microsoft.AspNetCore.Builder.UseExtensions.Use. The preceding `Use` extension method adds a middleware delegate defined in-line to the application's request pipeline.

There are two overloads available for the `Use` extension:

- One takes a HttpContext and a `Func<Task>`. Invoke the `Func<Task>` without any parameters.
- The other takes a `HttpContext` and a RequestDelegate. Invoke the `RequestDelegate` by passing the `HttpContext`.

Prefer using the later overload as it saves two internal per-request allocations that are required when using the other overload.

Test the middleware by passing in the culture. For example, request `https://localhost:5001/?culture=es-es`.

For ASP.NET Core's built-in localization support, see Globalization and localization in ASP.NET Core.

The following code moves the middleware delegate to a class:

```
C#
```

```
using System.Globalization;

namespace Middleware.Example;

public class RequestCultureMiddleware
{
    private readonly RequestDelegate _next;

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

    public async Task InvokeAsync(HttpContext context)
    {
        var cultureQuery = context.Request.Query["culture"];
        if (!string.IsNullOrWhiteSpace(cultureQuery))
        {
            var culture = new CultureInfo(cultureQuery);

            CultureInfo.CurrentCulture = culture;
            CultureInfo.CurrentUICulture = culture;
        }

        // Call the next delegate/middleware in the pipeline.
        await _next(context);
    }
}
```

The middleware class must include:

- A public constructor with a parameter of type RequestDelegate.
- A public method named `Invoke` or `InvokeAsync`. This method must:
  - Return a `Task`.
  - Accept a first parameter of type HttpContext.

Additional parameters for the constructor and `Invoke`/`InvokeAsync` are populated by dependency injection (DI).

Typically, an extension method is created to expose the middleware through IApplicationBuilder:

C#

```
using System.Globalization;

namespace Middleware.Example;

public class RequestCultureMiddleware
{
```

```
    private readonly RequestDelegate _next;

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

    public async Task InvokeAsync(HttpContext context)
    {
        var cultureQuery = context.Request.Query["culture"];
        if (!string.IsNullOrWhiteSpace(cultureQuery))
        {
            var culture = new CultureInfo(cultureQuery);

            CultureInfo.CurrentCulture = culture;
            CultureInfo.CurrentUICulture = culture;
        }

        // Call the next delegate/middleware in the pipeline.
        await _next(context);
    }
}

public static class RequestCultureMiddlewareExtensions
{
    public static IApplicationBuilder UseRequestCulture(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<RequestCultureMiddleware>();
    }
}
```

The following code calls the middleware from `Program.cs`:

C#

```
using Middleware.Example;
using System.Globalization;

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

app.UseHttpsRedirection();

app.UseRequestCulture();

app.Run(async (context) =>
{
    await context.Response.WriteAsync(
        $"CurrentCulture.DisplayName:
{CultureInfo.CurrentCulture.DisplayName}");
});
```

```
app.Run();
```

Middleware dependencies

Middleware should follow the Explicit Dependencies Principle by exposing its dependencies in its constructor. Middleware is constructed once per *application lifetime*.

Middleware components can resolve their dependencies from dependency injection (DI) through constructor parameters. UseMiddleware can also accept additional parameters directly.

Per-request middleware dependencies

Middleware is constructed at app startup and therefore has application life time. Scoped lifetime services used by middleware constructors aren't shared with other dependency-injected types during each request. To share a *scoped* service between middleware and other types, add these services to the `InvokeAsync` method's signature. The `InvokeAsync` method can accept additional parameters that are populated by DI:

C#

```
namespace Middleware.Example;

public class MyCustomMiddleware
{
    private readonly RequestDelegate _next;

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

    // IMessageWriter is injected into InvokeAsync
    public async Task InvokeAsync(HttpContext httpContext, IMessageWriter
svc)
    {
        svc.Write(DateTime.Now.Ticks.ToString());
        await _next(httpContext);
    }
}

public static class MyCustomMiddlewareExtensions
{
    public static IApplicationBuilder UseMyCustomMiddleware(
        this IApplicationBuilder builder)
    {
```

```
        return builder.UseMiddleware<MyCustomMiddleware>();
    }
}
```

Lifetime and registration options contains a complete sample of middleware with *scoped* lifetime services.

The following code is used to test the preceding middleware:

C#

```
using Middleware.Example;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IMessageWriter, LoggingMessageWriter>();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseMyCustomMiddleware();

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

app.Run();
```

The `IMessageWriter` interface and implementation:

C#

```
namespace Middleware.Example;

public interface IMessageWriter
{
    void Write(string message);
}

public class LoggingMessageWriter : IMessageWriter
{

    private readonly ILogger<LoggingMessageWriter> _logger;

    public LoggingMessageWriter(ILogger<LoggingMessageWriter> logger) =>
        _logger = logger;

    public void Write(string message) =>
        _logger.LogInformation(message);
}
```

Additional resources

- Sample code used in this article
- UseExtensions source on GitHub
- Lifetime and registration options contains a complete sample of middleware with *scoped*, *transient*, and *singleton* lifetime services.
- DEEP DIVE: HOW IS THE ASP.NET CORE MIDDLEWARE PIPELINE BUILT
- ASP.NET Core Middleware
- Test ASP.NET Core middleware
- Migrate HTTP handlers and modules to ASP.NET Core middleware
- App startup in ASP.NET Core
- Request Features in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Middleware activation with a third-party container in ASP.NET Core

Request and response operations in ASP.NET Core

Article • 07/26/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Justin Kotalik

This article explains how to read from the request body and write to the response body. Code for these operations might be required when writing middleware. Outside of writing middleware, custom code isn't generally required because the operations are handled by MVC and Razor Pages.

There are two abstractions for the request and response bodies: Stream and Pipe. For request reading, HttpRequest.Body is a Stream, and `HttpRequest.BodyReader` is a PipeReader. For response writing, HttpResponse.Body is a Stream, and `HttpResponse.BodyWriter` is a PipeWriter.

Pipelines are recommended over streams. Streams can be easier to use for some simple operations, but pipelines have a performance advantage and are easier to use in most scenarios. ASP.NET Core is starting to use pipelines instead of streams internally. Examples include:

- `FormReader`
- `TextReader`
- `TextWriter`
- `HttpResponse.WriteAsync`

Streams aren't being removed from the framework. Streams continue to be used throughout .NET, and many stream types don't have pipe equivalents, such as `FileStreams` and `ResponseCompression`.

Stream examples

Suppose the goal is to create a middleware that reads the entire request body as a list of strings, splitting on new lines. A simple stream implementation might look like the following example:

> **⚠ Warning**
>
> The following code:
>
> - Is used to demonstrate the problems with not using a pipe to read the request body.
> - Is not intended to be used in production apps.

C#

```
private async Task<List<string>> GetListOfStringsFromStream(Stream
requestBody)
{
    // Build up the request body in a string builder.
    StringBuilder builder = new StringBuilder();

    // Rent a shared buffer to write the request body into.
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);

    while (true)
    {
        var bytesRemaining = await requestBody.ReadAsync(buffer, offset: 0,
buffer.Length);
        if (bytesRemaining == 0)
        {
            break;
        }

        // Append the encoded string into the string builder.
        var encodedString = Encoding.UTF8.GetString(buffer, 0,
bytesRemaining);
        builder.Append(encodedString);
    }

    ArrayPool<byte>.Shared.Return(buffer);

    var entireRequestBody = builder.ToString();

    // Split on \n in the string.
    return new List<string>(entireRequestBody.Split("\n"));
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

This code works, but there are some issues:

- Before appending to the `StringBuilder`, the example creates another string (`encodedString`) that is thrown away immediately. This process occurs for all bytes in the stream, so the result is extra memory allocation the size of the entire request body.
- The example reads the entire string before splitting on new lines. It's more efficient to check for new lines in the byte array.

Here's an example that fixes some of the preceding issues:

> **⚠ Warning**
>
> The following code:
>
> - Is used to demonstrate the solutions to some problems in the preceding code while not solving all the problems.
> - Is not intended to be used in production apps.

C#

```
private async Task<List<string>>
GetListOfStringsFromStreamMoreEfficient(Stream requestBody)
{
    StringBuilder builder = new StringBuilder();
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);
    List<string> results = new List<string>();

    while (true)
    {
        var bytesRemaining = await requestBody.ReadAsync(buffer, offset: 0,
buffer.Length);

        if (bytesRemaining == 0)
        {
            results.Add(builder.ToString());
            break;
        }

        // Instead of adding the entire buffer into the StringBuilder
        // only add the remainder after the last \n in the array.
        var prevIndex = 0;
        int index;
        while (true)
        {
            index = Array.IndexOf(buffer, (byte)'\n', prevIndex);
            if (index == -1)
            {
```

```
                break;
            }

            var encodedString = Encoding.UTF8.GetString(buffer, prevIndex,
index - prevIndex);

            if (builder.Length > 0)
            {
                // If there was a remainder in the string buffer, include it
in the next string.
                results.Add(builder.Append(encodedString).ToString());
                builder.Clear();
            }
            else
            {
                results.Add(encodedString);
            }

            // Skip past last \n
            prevIndex = index + 1;
        }

        var remainingString = Encoding.UTF8.GetString(buffer, prevIndex,
bytesRemaining - prevIndex);
        builder.Append(remainingString);
    }

    ArrayPool<byte>.Shared.Return(buffer);

    return results;
}
```

This preceding example:

- Doesn't buffer the entire request body in a `StringBuilder` unless there aren't any newline characters.
- Doesn't call `Split` on the string.

However, there are still a few issues:

- If newline characters are sparse, much of the request body is buffered in the string.
- The code continues to create strings (`remainingString`) and adds them to the string buffer, which results in an extra allocation.

These issues are fixable, but the code is becoming progressively more complicated with little improvement. Pipelines provide a way to solve these problems with minimal code complexity.

Pipelines

The following example shows how the same scenario can be handled using a PipeReader:

C#

```
private async Task<List<string>> GetListOfStringFromPipe(PipeReader reader)
{
    List<string> results = new List<string>();

    while (true)
    {
        ReadResult readResult = await reader.ReadAsync();
        var buffer = readResult.Buffer;

        SequencePosition? position = null;

        do
        {
            // Look for a EOL in the buffer
            position = buffer.PositionOf((byte)'\n');

            if (position != null)
            {
                var readOnlySequence = buffer.Slice(0, position.Value);
                AddStringToList(results, in readOnlySequence);

                // Skip the line + the \n character (basically position)
                buffer = buffer.Slice(buffer.GetPosition(1,
position.Value));
            }
        }
        while (position != null);


        if (readResult.IsCompleted && buffer.Length > 0)
        {
            AddStringToList(results, in buffer);
        }

        reader.AdvanceTo(buffer.Start, buffer.End);

        // At this point, buffer will be updated to point one byte after the
last
        // \n character.
        if (readResult.IsCompleted)
        {
            break;
        }
    }

    return results;
}

private static void AddStringToList(List<string> results, in
```

```
ReadOnlySequence<byte> readOnlySequence)
{
    // Separate method because Span/ReadOnlySpan cannot be used in async
methods
    ReadOnlySpan<byte> span = readOnlySequence.IsSingleSegment ?
readOnlySequence.First.Span : readOnlySequence.ToArray().AsSpan();
    results.Add(Encoding.UTF8.GetString(span));
}
```

This example fixes many issues that the streams implementations had:

- There's no need for a string buffer because the `PipeReader` handles bytes that haven't been used.
- Encoded strings are directly added to the list of returned strings.
- Other than the `ToArray` call, and the memory used by the string, string creation is allocation free.

Adapters

The `Body`, `BodyReader`, and `BodyWriter` properties are available for `HttpRequest` and `HttpResponse`. When you set `Body` to a different stream, a new set of adapters automatically adapt each type to the other. If you set `HttpRequest.Body` to a new stream, `HttpRequest.BodyReader` is automatically set to a new `PipeReader` that wraps `HttpRequest.Body`.

StartAsync

`HttpResponse.StartAsync` is used to indicate that headers are unmodifiable and to run `OnStarting` callbacks. When using Kestrel as a server, calling `StartAsync` before using the `PipeReader` guarantees that memory returned by `GetMemory` belongs to Kestrel's internal Pipe rather than an external buffer.

Additional resources

- System.IO.Pipelines in .NET
- Write custom ASP.NET Core middleware

Request decompression in ASP.NET Core

Article • 09/27/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By David Acker

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.

When the `Content-Encoding` header value on a request matches one of the available decompression providers, the middleware:

- Uses the matching provider to wrap the HttpRequest.Body in an appropriate decompression stream.
- Removes the `Content-Encoding` header, indicating that the request body is no longer compressed.

Requests that don't include a `Content-Encoding` header are ignored by the request decompression middleware.

Decompression:

- Occurs when the body of the request is read. That is, decompression occurs at the endpoint on model binding. The request body isn't decompressed eagerly.
- When attempting to read the decompressed request body with invalid compressed data for the specified `Content-Encoding`, an exception is thrown. Brotli can throw System.InvalidOperationException: Decoder ran into invalid data. Deflate and GZip can throw System.IO.InvalidDataException: The archive entry was compressed using an unsupported compression method.

If the middleware encounters a request with compressed content but is unable to decompress it, the request is passed to the next delegate in the pipeline. For example, a request with an unsupported `Content-Encoding` header value or multiple `Content-Encoding` header values is passed to the next delegate in the pipeline.

Configuration

The following code uses AddRequestDecompression(IServiceCollection) and UseRequestDecompression to enable request decompression for the default `Content-Encoding` types:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRequestDecompression();

var app = builder.Build();

app.UseRequestDecompression();

app.MapPost("/", (HttpRequest request) => Results.Stream(request.Body));

app.Run();
```

Default decompression providers

The `Content-Encoding` header values that the request decompression middleware supports by default are listed in the following table:

Expand table

Content-Encoding header values	Description
`br`	Brotli compressed data format
`deflate`	DEFLATE compressed data format
`gzip`	Gzip file format

Custom decompression providers

Support for custom encodings can be added by creating custom decompression provider classes that implement IDecompressionProvider:

```
C#

public class CustomDecompressionProvider : IDecompressionProvider
{
    public Stream GetDecompressionStream(Stream stream)
    {
        // Perform custom decompression logic here
        return stream;
    }
}
```

Custom decompression providers are registered with RequestDecompressionOptions along with their corresponding `Content-Encoding` header values:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRequestDecompression(options =>
{
    options.DecompressionProviders.Add("custom", new
CustomDecompressionProvider());
});

var app = builder.Build();

app.UseRequestDecompression();

app.MapPost("/", (HttpRequest request) => Results.Stream(request.Body));

app.Run();
```

Request size limits

In order to protect against zip bombs or decompression bombs:

- The maximum size of the decompressed request body is limited to the request body size limit enforced by the endpoint or server.
- If the number of bytes read from the decompressed request body stream exceeds the limit, an InvalidOperationException is thrown to prevent additional bytes from being read from the stream.

In order of precedence, the maximum request size for an endpoint is set by:

1. IRequestSizeLimitMetadata.MaxRequestBodySize, such as RequestSizeLimitAttribute or DisableRequestSizeLimitAttribute for MVC endpoints.
2. The global server size limit IHttpMaxRequestBodySizeFeature.MaxRequestBodySize. `MaxRequestBodySize` can be overridden per request with IHttpMaxRequestBodySizeFeature.MaxRequestBodySize, but defaults to the limit configured for the web server implementation.

Expand table

Web server implementation	`MaxRequestBodySize` configuration
HTTP.sys	HttpSysOptions.MaxRequestBodySize
IIS	IISServerOptions.MaxRequestBodySize
Kestrel	KestrelServerLimits.MaxRequestBodySize

> ⚠ **Warning**
>
> Disabling the request body size limit poses a security risk in regards to uncontrolled resource consumption, particularly if the request body is being buffered. Ensure that safeguards are in place to mitigate the risk of denial-of-service (DoS) attacks.

Additional Resources

- ASP.NET Core Middleware
- Mozilla Developer Network: Content-Encoding
- Brotli Compressed Data Format
- DEFLATE Compressed Data Format Specification version 1.3
- GZIP file format specification version 4.3

Factory-based middleware activation in ASP.NET Core

Article • 07/26/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

IMiddlewareFactory/IMiddleware is an extensibility point for middleware activation that offers the following benefits:

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

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

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

IMiddleware

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

Middleware activated by convention:

C#

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

    public ConventionalMiddleware(RequestDelegate next)
```

```
        => _next = next;

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

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

            await dbContext.SaveChangesAsync();
        }

        await _next(context);
    }
}
```

Middleware activated by MiddlewareFactory:

C#

```
public class FactoryActivatedMiddleware : IMiddleware
{
    private readonly SampleDbContext _dbContext;

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

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

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

            await _dbContext.SaveChangesAsync();
        }

        await next(context);
    }
}
```

Extensions are created for the middleware:

C#

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

```
        => app.UseMiddleware<ConventionalMiddleware>();

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

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

C#

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

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

C#

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

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

builder.Services.AddTransient<FactoryActivatedMiddleware>();
```

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

C#

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

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

IMiddlewareFactory

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

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

Additional resources

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

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

Article • 07/26/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

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

View or download sample code (how to download)

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

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

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

IMiddlewareFactory

IMiddlewareFactory provides methods to create middleware.

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

C#

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

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

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

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

IMiddleware

IMiddleware defines middleware for the app's request pipeline.

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

C#

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

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

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

        if (!string.IsNullOrWhiteSpace(keyValue))
```

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

            await _db.SaveChangesAsync();
        }

        await next(context);
    }
}
```

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

C#

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

`Startup.ConfigureServices` must perform several tasks:

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

C#

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

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

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

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

    _container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

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

    _container.Register<SimpleInjectorActivatedMiddleware>();

    _container.Verify();
}
```

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

C#

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

    app.UseSimpleInjectorActivatedMiddleware();

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

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

Additional resources

- Middleware

- Factory-based middleware activation
- Simple Injector GitHub repository
- Simple Injector documentation

WebApplication and WebApplicationBuilder in Minimal API apps

Article • 07/26/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

`WebApplication`

The following code is generated by an ASP.NET Core template:

```
C#

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

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

app.Run();
```

The preceding code can be created via `dotnet new web` on the command line or selecting the Empty Web template in Visual Studio.

The following code creates a WebApplication (`app`) without explicitly creating a WebApplicationBuilder:

```
C#

var app = WebApplication.Create(args);

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

app.Run();
```

WebApplication.Create initializes a new instance of the WebApplication class with preconfigured defaults.

Working with ports

When a web app is created with Visual Studio or `dotnet new`, a `Properties/launchSettings.json` file is created that specifies the ports the app responds to. In the port setting samples that follow, running the app from Visual Studio returns an error dialog `Unable to connect to web server 'AppName'`. Visual Studio returns an error because it's expecting the port specified in `Properties/launchSettings.json`, but the app is using the port specified by `app.Run("http://localhost:3000")`. Run the following port changing samples from the command line.

The following sections set the port the app responds to.

C#

```
var app = WebApplication.Create(args);

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

app.Run("http://localhost:3000");
```

In the preceding code, the app responds to port `3000`.

Multiple ports

In the following code, the app responds to port `3000` and `4000`.

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://localhost:3000");
app.Urls.Add("http://localhost:4000");

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

app.Run();
```

Set the port from the command line

The following command makes the app respond to port `7777`:

.NET CLI

```
dotnet run --urls="https://localhost:7777"
```

If the Kestrel endpoint is also configured in the `appsettings.json` file, the `appsettings.json` file specified URL is used. For more information, see Kestrel endpoint configuration

Read the port from environment

The following code reads the port from the environment:

C#

```
var app = WebApplication.Create(args);

var port = Environment.GetEnvironmentVariable("PORT") ?? "3000";

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

app.Run($"http://localhost:{port}");
```

The preferred way to set the port from the environment is to use the `ASPNETCORE_URLS` environment variable, which is shown in the following section.

Set the ports via the ASPNETCORE_URLS environment variable

The `ASPNETCORE_URLS` environment variable is available to set the port:

```
ASPNETCORE_URLS=http://localhost:3000
```

`ASPNETCORE_URLS` supports multiple URLs:

```
ASPNETCORE_URLS=http://localhost:3000;https://localhost:5000
```

Listen on all interfaces

The following samples demonstrate listening on all interfaces

http://*:3000

```
C#

var app = WebApplication.Create(args);

app.Urls.Add("http://*:3000");

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

app.Run();
```

http://+:3000

```
C#

var app = WebApplication.Create(args);

app.Urls.Add("http://+:3000");

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

app.Run();
```

`http://0.0.0.0:3000`

```
C#

var app = WebApplication.Create(args);

app.Urls.Add("http://0.0.0.0:3000");

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

app.Run();
```

Listen on all interfaces using ASPNETCORE_URLS

The preceding samples can use `ASPNETCORE_URLS`

```

ASPNETCORE_URLS=http://*:3000;https://+:5000;http://0.0.0.0:5005
```

Listen on all interfaces using ASPNETCORE_HTTPS_PORTS

The preceding samples can use `ASPNETCORE_HTTPS_PORTS` and `ASPNETCORE_HTTP_PORTS`.

```
ASPNETCORE_HTTP_PORTS=3000;5005
ASPNETCORE_HTTPS_PORTS=5000
```

For more information, see Configure endpoints for the ASP.NET Core Kestrel web server

Specify HTTPS with development certificate

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("https://localhost:3000");

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

app.Run();
```

For more information on the development certificate, see Trust the ASP.NET Core HTTPS development certificate on Windows and macOS.

Specify HTTPS using a custom certificate

The following sections show how to specify the custom certificate using the `appsettings.json` file and via configuration.

Specify the custom certificate with `appsettings.json`

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
```

```
  "Kestrel": {
    "Certificates": {
      "Default": {
        "Path": "cert.pem",
        "KeyPath": "key.pem"
      }
    }
  }
}
```

Specify the custom certificate via configuration

C#

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

// Configure the cert and the key
builder.Configuration["Kestrel:Certificates:Default:Path"] = "cert.pem";
builder.Configuration["Kestrel:Certificates:Default:KeyPath"] = "key.pem";

var app = builder.Build();

app.Urls.Add("https://localhost:3000");

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

app.Run();
```

Use the certificate APIs

C#

```
using System.Security.Cryptography.X509Certificates;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(options =>
{
    options.ConfigureHttpsDefaults(httpsOptions =>
    {
        var certPath = Path.Combine(builder.Environment.ContentRootPath,
"cert.pem");
        var keyPath = Path.Combine(builder.Environment.ContentRootPath,
"key.pem");

        httpsOptions.ServerCertificate =
X509Certificate2.CreateFromPemFile(certPath,
                                           keyPath);
    });
```

```
});

var app = builder.Build();

app.Urls.Add("https://localhost:3000");

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

app.Run();
```

Read the environment

C#

```
var app = WebApplication.Create(args);

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

app.MapGet("/", () => "Hello World");
app.MapGet("/oops", () => "Oops! An error happened.");

app.Run();
```

For more information using the environment, see Use multiple environments in ASP.NET Core

Configuration

The following code reads from the configuration system:

C#

```
var app = WebApplication.Create(args);

var message = app.Configuration["HelloKey"] ?? "Config failed!";

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

app.Run();
```

For more information, see Configuration in ASP.NET Core

Logging

The following code writes a message to the log on application startup:

```
C#
```

```
var app = WebApplication.Create(args);

app.Logger.LogInformation("The app started");

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

app.Run();
```

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

Access the Dependency Injection (DI) container

The following code shows how to get services from the DI container during application startup:

```
C#
```

```


var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddScoped<SampleService>();

var app = builder.Build();

app.MapControllers();

using (var scope = app.Services.CreateScope())
{
    var sampleService =
scope.ServiceProvider.GetRequiredService<SampleService>();
    sampleService.DoSomething();
}

app.Run();
```

The following code shows how to access keys from the DI container using the [FromKeyedServices] attribute:

```
C#
```

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

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
```

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

app.MapGet("/big", ([FromKeyedServices("big")] ICache bigCache) =>
bigCache.Get("date"));

app.MapGet("/small", ([FromKeyedServices("small")] ICache smallCache) =>
smallCache.Get("date"));

app.Run();

public interface ICache
{
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}
```

For more information on DI, see Dependency injection in ASP.NET Core.

WebApplicationBuilder

This section contains sample code using WebApplicationBuilder.

Change the content root, application name, and environment

The following code sets the content root, application name, and environment:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    ApplicationName = typeof(Program).Assembly.FullName,
    ContentRootPath = Directory.GetCurrentDirectory(),
    EnvironmentName = Environments.Staging,
    WebRootPath = "customwwwroot"
});

Console.WriteLine($"Application Name:
{builder.Environment.ApplicationName}");
```

```
Console.WriteLine($"Environment Name:
{builder.Environment.EnvironmentName}");
Console.WriteLine($"ContentRoot Path:
{builder.Environment.ContentRootPath}");
Console.WriteLine($"WebRootPath: {builder.Environment.WebRootPath}");

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

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults.

For more information, see ASP.NET Core fundamentals overview

Change the content root, app name, and environment by using environment variables or command line

The following table shows the environment variable and command-line argument used to change the content root, app name, and environment:

Expand table

feature	Environment variable	Command-line argument
Application name	ASPNETCORE_APPLICATIONNAME	--applicationName
Environment name	ASPNETCORE_ENVIRONMENT	--environment
Content root	ASPNETCORE_CONTENTROOT	--contentRoot

Add configuration providers

The following sample adds the INI configuration provider:

C#

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

builder.Configuration.AddIniFile("appsettings.ini");

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

For detailed information, see File configuration providers in Configuration in ASP.NET Core.

Read configuration

By default the WebApplicationBuilder reads configuration from multiple sources, including:

- `appSettings.json` and `appSettings.{environment}.json`
- Environment variables
- The command line

For a complete list of configuration sources read, see Default configuration in Configuration in ASP.NET Core.

The following code reads `HelloKey` from configuration and displays the value at the `/` endpoint. If the configuration value is null, "Hello" is assigned to `message`:

C#

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

var message = builder.Configuration["HelloKey"] ?? "Hello";

var app = builder.Build();

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

app.Run();
```

Read the environment

C#

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

if (builder.Environment.IsDevelopment())
{
    Console.WriteLine($"Running in development.");
}

var app = builder.Build();

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

app.Run();
```

Add logging providers

C#

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

// Configure JSON logging to the console.
builder.Logging.AddJsonConsole();

var app = builder.Build();

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

app.Run();
```

Add services

C#

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

// Add the memory cache services.
builder.Services.AddMemoryCache();

// Add a custom scoped service.
builder.Services.AddScoped<ITodoRepository, TodoRepository>();
var app = builder.Build();
```

Customize the IHostBuilder

Existing extension methods on IHostBuilder can be accessed using the Host property:

C#

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

// Wait 30 seconds for graceful shutdown.
builder.Host.ConfigureHostOptions(o => o.ShutdownTimeout =
TimeSpan.FromSeconds(30));

var app = builder.Build();

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

app.Run();
```

Customize the IWebHostBuilder

Extension methods on IWebHostBuilder can be accessed using the WebApplicationBuilder.WebHost property.

C#

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

// Change the HTTP server implemenation to be HTTP.sys based
builder.WebHost.UseHttpSys();

var app = builder.Build();

app.MapGet("/", () => "Hello HTTP.sys");

app.Run();
```

Change the web root

By default, the web root is relative to the content root in the `wwwroot` folder. Web root is where the static files middleware looks for static files. Web root can be changed with `WebHostOptions`, the command line, or with the UseWebRoot method:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    // Look for static files in webroot
    WebRootPath = "webroot"
});

var app = builder.Build();

app.Run();
```

Custom dependency injection (DI) container

The following example uses Autofac:

C#

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

builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());

// Register services directly with Autofac here. Don't
// call builder.Populate(), that happens in AutofacServiceProviderFactory.
builder.Host.ConfigureContainer<ContainerBuilder>(builder =>
builder.RegisterModule(new MyApplicationModule()));
```

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

Add Middleware

Any existing ASP.NET Core middleware can be configured on the `WebApplication`:

C#

```
var app = WebApplication.Create(args);

// Setup the file server to serve static files.
app.UseFileServer();

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

app.Run();
```

For more information, see ASP.NET Core Middleware

Developer exception page

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults. The developer exception page is enabled in the preconfigured defaults. When the following code is run in the development environment, navigating to `/` renders a friendly page that shows the exception.

C#

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

var app = builder.Build();

app.MapGet("/", () =>
{
    throw new InvalidOperationException("Oops, the '/' route has thrown an
exception.");
});

app.Run();
```

.NET Generic Host in ASP.NET Core

Article • 09/10/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This article provides information on using the .NET Generic Host in ASP.NET Core.

The ASP.NET Core templates create a WebApplicationBuilder and WebApplication, which provide a streamlined way to configure and run web applications without a `Startup` class. For more information on `WebApplicationBuilder` and `WebApplication`, see Migrate from ASP.NET Core 5.0 to 6.0.

For information on using the .NET Generic Host in console apps, see .NET Generic Host.

Host definition

A *host* is an object that encapsulates an app's resources, such as:

- Dependency injection (DI)
- Logging
- Configuration
- `IHostedService` implementations

When a host starts, it calls IHostedService.StartAsync on each implementation of IHostedService registered in the service container's collection of hosted services. In a web app, one of the `IHostedService` implementations is a web service that starts an HTTP server implementation.

Including all of the app's interdependent resources in one object enables control over app startup and graceful shutdown.

Set up a host

The host is typically configured, built, and run by code in the `Program.cs`. The following code creates a host with an `IHostedService` implementation added to the DI container:

C#

```
await Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddHostedService<SampleHostedService>();
    })
    .Build()
    .RunAsync();
```

For an HTTP workload, call ConfigureWebHostDefaults after CreateDefaultBuilder:

C#

```
await Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    })
    .Build()
    .RunAsync();
```

Default builder settings

The CreateDefaultBuilder method:

- Sets the content root to the path returned by GetCurrentDirectory.
- Loads host configuration from:
  - Environment variables prefixed with `DOTNET_`.
  - Command-line arguments.
- Loads app configuration from:
  - `appsettings.json`.
  - `appsettings.{Environment}.json`.
  - User secrets when the app runs in the `Development` environment.
  - Environment variables.
  - Command-line arguments.
- Adds the following logging providers:
  - Console
  - Debug
  - EventSource
  - EventLog (only when running on Windows)

- Enables scope validation and dependency validation when the environment is Development.

The ConfigureWebHostDefaults method:

- Loads host configuration from environment variables prefixed with `ASPNETCORE_`.
- Sets Kestrel server as the web server and configures it using the app's hosting configuration providers. For the Kestrel server's default options, see Configure options for the ASP.NET Core Kestrel web server.
- Adds Host Filtering middleware.
- Adds Forwarded Headers middleware if `ASPNETCORE_FORWARDEDHEADERS_ENABLED` equals `true`.
- Enables IIS integration. For the IIS default options, see Host ASP.NET Core on Windows with IIS.

The Settings for all app types and Settings for web apps sections later in this article show how to override default builder settings.

Framework-provided services

The following services are registered automatically:

- IHostApplicationLifetime
- IHostLifetime
- IHostEnvironment / IWebHostEnvironment

For more information on framework-provided services, see Dependency injection in ASP.NET Core.

IHostApplicationLifetime

Inject the IHostApplicationLifetime (formerly `IApplicationLifetime`) service into any class to handle post-startup and graceful shutdown tasks. Three properties on the interface are cancellation tokens used to register app start and app stop event handler methods. The interface also includes a `StopApplication` method, which allows apps to request a graceful shutdown.

When performing a graceful shutdown, the host:

- Triggers the ApplicationStopping event handlers, which allows the app to run logic before the shutdown process begins.

- Stops the server, which disables new connections. The server waits for requests on existing connections to complete, for as long as the shutdown timeout allows. The server sends the connection close header for further requests on existing connections.
- Triggers the ApplicationStopped event handlers, which allows the app to run logic after the application has shutdown.

The following example is an `IHostedService` implementation that registers `IHostApplicationLifetime` event handlers:

C#

```
public class HostApplicationLifetimeEventsHostedService : IHostedService
{
    private readonly IHostApplicationLifetime _hostApplicationLifetime;

    public HostApplicationLifetimeEventsHostedService(
        IHostApplicationLifetime hostApplicationLifetime)
        => _hostApplicationLifetime = hostApplicationLifetime;

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _hostApplicationLifetime.ApplicationStarted.Register(OnStarted);
        _hostApplicationLifetime.ApplicationStopping.Register(OnStopping);
        _hostApplicationLifetime.ApplicationStopped.Register(OnStopped);

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
        => Task.CompletedTask;

    private void OnStarted()
    {
        // ...
    }

    private void OnStopping()
    {
        // ...
    }

    private void OnStopped()
    {
        // ...
    }
}
```

IHostLifetime

The IHostLifetime implementation controls when the host starts and when it stops. The last implementation registered is used.

`Microsoft.Extensions.Hosting.Internal.ConsoleLifetime` is the default `IHostLifetime` implementation. `ConsoleLifetime`:

- Listens for `Ctrl`+`C`/SIGINT (Windows), `⌘`+`C` (macOS), or SIGTERM and calls StopApplication to start the shutdown process.
- Unblocks extensions such as RunAsync and WaitForShutdownAsync.

IHostEnvironment

Inject the IHostEnvironment service into a class to get information about the following settings:

- ApplicationName
- EnvironmentName
- ContentRootPath

Web apps implement the `IWebHostEnvironment` interface, which inherits `IHostEnvironment` and adds the WebRootPath.

Host configuration

Host configuration is used for the properties of the IHostEnvironment implementation.

Host configuration is available from HostBuilderContext.Configuration inside ConfigureAppConfiguration. After `ConfigureAppConfiguration`, `HostBuilderContext.Configuration` is replaced with the app config.

To add host configuration, call ConfigureHostConfiguration on `IHostBuilder`. `ConfigureHostConfiguration` can be called multiple times with additive results. The host uses whichever option sets a value last on a given key.

The environment variable provider with prefix `DOTNET_` and command-line arguments are included by `CreateDefaultBuilder`. For web apps, the environment variable provider with prefix `ASPNETCORE_` is added. The prefix is removed when the environment variables are read. For example, the environment variable value for `ASPNETCORE_ENVIRONMENT` becomes the host configuration value for the `environment` key.

The following example creates host configuration:

```
```

C#

```
Host.CreateDefaultBuilder(args)
    .ConfigureHostConfiguration(hostConfig =>
    {
        hostConfig.SetBasePath(Directory.GetCurrentDirectory());
        hostConfig.AddJsonFile("hostsettings.json", optional: true);
        hostConfig.AddEnvironmentVariables(prefix: "PREFIX_");
        hostConfig.AddCommandLine(args);
    });
```

App configuration

App configuration is created by calling ConfigureAppConfiguration on `IHostBuilder`. `ConfigureAppConfiguration` can be called multiple times with additive results. The app uses whichever option sets a value last on a given key.

The configuration created by `ConfigureAppConfiguration` is available at HostBuilderContext.Configuration for subsequent operations and as a service from DI. The host configuration is also added to the app configuration.

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

Settings for all app types

This section lists host settings that apply to both HTTP and non-HTTP workloads. By default, environment variables used to configure these settings can have a `DOTNET_` or `ASPNETCORE_` prefix, which appear in the following list of settings as the `{PREFIX_}` placeholder. For more information, see the Default builder settings section and Configuration: Environment variables.

ApplicationName

The IHostEnvironment.ApplicationName property is set from host configuration during host construction.

Key: `applicationName`
Type: `string`
Default: The name of the assembly that contains the app's entry point.
Environment variable: `{PREFIX_}APPLICATIONNAME`

To set this value, use the environment variable.

ContentRoot

The IHostEnvironment.ContentRootPath property determines where the host begins searching for content files. If the path doesn't exist, the host fails to start.

Key: `contentRoot`
Type: `string`
Default: The folder where the app assembly resides.
Environment variable: `{PREFIX_}CONTENTROOT`

To set this value, use the environment variable or call `UseContentRoot` on `IHostBuilder`:

```
C#

Host.CreateDefaultBuilder(args)
    .UseContentRoot("/path/to/content/root")
    // ...
```

For more information, see:

- Fundamentals: Content root
- WebRoot

EnvironmentName

The IHostEnvironment.EnvironmentName property can be set to any value. Framework-defined values include `Development`, `Staging`, and `Production`. Values aren't case-sensitive.

Key: `environment`
Type: `string`
Default: `Production`
Environment variable: `{PREFIX_}ENVIRONMENT`

To set this value, use the environment variable or call `UseEnvironment` on `IHostBuilder`:

```
C#

Host.CreateDefaultBuilder(args)
    .UseEnvironment("Development")
    // ...
```

ShutdownTimeout

HostOptions.ShutdownTimeout sets the timeout for StopAsync. The default value is 30 seconds. During the timeout period, the host:

- Triggers IHostApplicationLifetime.ApplicationStopping.
- Attempts to stop hosted services, logging errors for services that fail to stop.

If the timeout period expires before all of the hosted services stop, any remaining active services are stopped when the app shuts down. The services stop even if they haven't finished processing. If services require more time to stop, increase the timeout.

Key: `shutdownTimeoutSeconds`
Type: `int`
Default: 30 seconds
Environment variable: `{PREFIX_}SHUTDOWNTIMEOUTSECONDS`

To set this value, use the environment variable or configure `HostOptions`. The following example sets the timeout to 20 seconds:

C#

```
Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) =>
    {
        services.Configure<HostOptions>(options =>
        {
            options.ShutdownTimeout = TimeSpan.FromSeconds(20);
        });
    });
```

Disable app configuration reload on change

By default, `appsettings.json` and `appsettings.{Environment}.json` are reloaded when the file changes. To disable this reload behavior in ASP.NET Core 5.0 or later, set the `hostBuilder:reloadConfigOnChange` key to `false`.

Key: `hostBuilder:reloadConfigOnChange`
Type: `bool` (`true` or `false`)
Default: `true`
Command-line argument: `hostBuilder:reloadConfigOnChange`
Environment variable: `{PREFIX_}hostBuilder:reloadConfigOnChange`

> ⚠ **Warning**

The colon (`:`) separator doesn't work with environment variable hierarchical keys on all platforms. For more information, see **Environment variables**.

Settings for web apps

Some host settings apply only to HTTP workloads. By default, environment variables used to configure these settings can have a `DOTNET_` or `ASPNETCORE_` prefix, which appear in the following list of settings as the `{PREFIX_}` placeholder.

Extension methods on `IWebHostBuilder` are available for these settings. Code samples that show how to call the extension methods assume `webBuilder` is an instance of `IWebHostBuilder`, as in the following example:

C#

```
Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
        // ...
    });
```

CaptureStartupErrors

When `false`, errors during startup result in the host exiting. When `true`, the host captures exceptions during startup and attempts to start the server.

Key: `captureStartupErrors`
Type: `bool` (`true`/`1` or `false`/`0`)
Default: Defaults to `false` unless the app runs with Kestrel behind IIS, where the default is `true`.
Environment variable: `{PREFIX_}CAPTURESTARTUPERRORS`

To set this value, use configuration or call `CaptureStartupErrors`:

C#

```
webBuilder.CaptureStartupErrors(true);
```

DetailedErrors

When enabled, or when the environment is `Development`, the app captures detailed errors.

Key: `detailedErrors`
Type: `bool` (`true`/`1` or `false`/`0`)
Default: `false`
Environment variable: `{PREFIX_}DETAILEDERRORS`

To set this value, use configuration or call `UseSetting`:

```
C#

webBuilder.UseSetting(WebHostDefaults.DetailedErrorsKey, "true");
```

HostingStartupAssemblies

A semicolon-delimited string of hosting startup assemblies to load on startup. Although the configuration value defaults to an empty string, the hosting startup assemblies always include the app's assembly. When hosting startup assemblies are provided, they're added to the app's assembly for loading when the app builds its common services during startup.

Key: `hostingStartupAssemblies`
Type: `string`
Default: Empty string
Environment variable: `{PREFIX_}HOSTINGSTARTUPASSEMBLIES`

To set this value, use configuration or call `UseSetting`:

```
C#

webBuilder.UseSetting(
    WebHostDefaults.HostingStartupAssembliesKey, "assembly1;assembly2");
```

HostingStartupExcludeAssemblies

A semicolon-delimited string of hosting startup assemblies to exclude on startup.

Key: `hostingStartupExcludeAssemblies`
Type: `string`
Default: Empty string
Environment variable: `{PREFIX_}HOSTINGSTARTUPEXCLUDEASSEMBLIES`

To set this value, use configuration or call `UseSetting`:

```
C#

webBuilder.UseSetting(
    WebHostDefaults.HostingStartupExcludeAssembliesKey,
"assembly1;assembly2");
```

HTTPS_Port

Set the HTTPS port to redirect to if you get a non-HTTPS connection. Used in enforcing HTTPS. This setting doesn't cause the server to listen on the specified port. That is, it's possible to accidentally redirect requests to an unused port.

Key: `https_port` **Type**: `string`
Default: A default value isn't set.
Environment variable: `{PREFIX_}HTTPS_PORT`

To set this value, use configuration or call `UseSetting`:

```
C#

webBuilder.UseSetting("https_port", "8080");
```

HTTPS_Ports

The ports to listen on for HTTPS connections.

Key: `https_ports`
Type: `string`
Default: A default value isn't set.
Environment variable: `{PREFIX_}HTTPS_PORTS`

To set this value, use configuration or call `UseSetting`:

```
C#

webBuilder.UseSetting("https_ports", "8080");
```

PreferHostingUrls

Indicates whether the host should listen on the URLs configured with the `IWebHostBuilder` instead of those URLs configured with the `IServer` implementation.

Key: `preferHostingUrls`
Type: `bool` (`true`/`1` or `false`/`0`)
Default: `false`
Environment variable: `{PREFIX_}PREFERHOSTINGURLS`

To set this value, use the environment variable or call `PreferHostingUrls`:

```
C#

webBuilder.PreferHostingUrls(true);
```

PreventHostingStartup

Prevents the automatic loading of hosting startup assemblies, including hosting startup assemblies configured by the app's assembly. For more information, see Use hosting startup assemblies in ASP.NET Core.

Key: `preventHostingStartup`
Type: `bool` (`true`/`1` or `false`/`0`)
Default: `false`
Environment variable: `{PREFIX_}PREVENTHOSTINGSTARTUP`

To set this value, use the environment variable or call `UseSetting` :

```
C#

webBuilder.UseSetting(WebHostDefaults.PreventHostingStartupKey, "true");
```

StartupAssembly

The assembly to search for the `Startup` class.

Key: `startupAssembly`
Type: `string`
Default: The app's assembly
Environment variable: `{PREFIX_}STARTUPASSEMBLY`

To set this value, use the environment variable or call `UseStartup`. `UseStartup` can take an assembly name (`string`) or a type (`TStartup`). If multiple `UseStartup` methods are called, the last one takes precedence.

```
C#

webBuilder.UseStartup("StartupAssemblyName");
```

```
C#

webBuilder.UseStartup<Startup>();
```

SuppressStatusMessages

When enabled, suppresses hosting startup status messages.

Key: `suppressStatusMessages`
Type: `bool` (`true`/`1` or `false`/`0`)
Default: `false`
Environment variable: `{PREFIX_}SUPPRESSSTATUSMESSAGES`

To set this value, use configuration or call `UseSetting`:

```
C#

webBuilder.UseSetting(WebHostDefaults.SuppressStatusMessagesKey, "true");
```

URLs

A semicolon-delimited list of IP addresses or host addresses with ports and protocols that the server should listen on for requests. For example, `http://localhost:123`. Use "*" to indicate that the server should listen for requests on any IP address or hostname using the specified port and protocol (for example, `http://*:5000`). The protocol (`http://` or `https://`) must be included with each URL. Supported formats vary among servers.

Key: `urls`
Type: `string`
Default: `http://localhost:5000` and `https://localhost:5001`
Environment variable: `{PREFIX_}URLS`

To set this value, use the environment variable or call `UseUrls`:

```
C#

webBuilder.UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002");
```

Kestrel has its own endpoint configuration API. For more information, see Configure endpoints for the ASP.NET Core Kestrel web server.

WebRoot

The IWebHostEnvironment.WebRootPath property determines the relative path to the app's static assets. If the path doesn't exist, a no-op file provider is used.

Key: `webroot`
Type: `string`
Default: The default is `wwwroot`. The path to *{content root}/wwwroot* must exist.
Environment variable: `{PREFIX_}WEBROOT`

To set this value, use the environment variable or call `UseWebRoot` on `IWebHostBuilder`:

```
C#

webBuilder.UseWebRoot("public");
```

For more information, see:

- Fundamentals: Web root
- ContentRoot

Manage the host lifetime

Call methods on the built IHost implementation to start and stop the app. These methods affect all IHostedService implementations that are registered in the service container.

The difference between `Run*` and `Start*` methods is that `Run*` methods wait for the host to complete before returning, whereas `Start*` methods return immediately. The `Run*` methods are typically used in console apps, whereas the `Start*` methods are typically used in long-running services.

Run

Run runs the app and blocks the calling thread until the host is shut down.

RunAsync

RunAsync runs the app and returns a Task that completes when the cancellation token or shutdown is triggered.

RunConsoleAsync

RunConsoleAsync enables console support, builds and starts the host, and waits for Ctrl+C/SIGINT (Windows), ⌘+C (macOS), or SIGTERM to shut down.

Start

Start starts the host synchronously.

StartAsync

StartAsync starts the host and returns a Task that completes when the cancellation token or shutdown is triggered.

WaitForStartAsync is called at the start of `StartAsync`, which waits until it's complete before continuing. This method can be used to delay startup until signaled by an external event.

StopAsync

StopAsync attempts to stop the host within the provided timeout.

WaitForShutdown

WaitForShutdown blocks the calling thread until shutdown is triggered by the IHostLifetime, such as via Ctrl+C/SIGINT (Windows), ⌘+C (macOS), or SIGTERM.

WaitForShutdownAsync

WaitForShutdownAsync returns a Task that completes when shutdown is triggered via the given token and calls StopAsync.

Additional resources

- Background tasks with hosted services in ASP.NET Core
- GitHub link to Generic Host source

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

ASP.NET Core Web Host

Article • 09/10/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

ASP.NET Core apps configure and launch a *host*. The host is responsible for app startup and lifetime management. At a minimum, the host configures a server and a request processing pipeline. The host can also set up logging, dependency injection, and configuration.

This article covers the Web Host, which remains available only for backward compatibility. The ASP.NET Core templates create a WebApplicationBuilder and WebApplication, which is recommended for web apps. For more information on `WebApplicationBuilder` and `WebApplication`, see Migrate from ASP.NET Core 5.0 to 6.0

Set up a host

Create a host using an instance of IWebHostBuilder. This is typically performed in the app's entry point, the `Main` method in `Program.cs`. A typical app calls CreateDefaultBuilder to start setting up a host:

C#

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

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

The code that calls `CreateDefaultBuilder` is in a method named `CreateWebHostBuilder`, which separates it from the code in `Main` that calls `Run` on the builder object. This separation is required if you use Entity Framework Core tools. The tools expect to find a `CreateWebHostBuilder` method that they can call at design time to configure the host without running the app. An alternative is to implement `IDesignTimeDbContextFactory`. For more information, see Design-time DbContext Creation.

`CreateDefaultBuilder` performs the following tasks:

- Configures Kestrel server as the web server using the app's hosting configuration providers. For the Kestrel server's default options, see Configure options for the ASP.NET Core Kestrel web server.
- Sets the content root to the path returned by Directory.GetCurrentDirectory.
- Loads host configuration from:
  - Environment variables prefixed with `ASPNETCORE_` (for example, `ASPNETCORE_ENVIRONMENT`).
  - Command-line arguments.
- Loads app configuration in the following order from:
  - `appsettings.json`.
  - `appsettings.{Environment}.json`.
  - User secrets when the app runs in the `Development` environment using the entry assembly.
  - Environment variables.
  - Command-line arguments.
- Configures logging for console and debug output. Logging includes log filtering rules specified in a Logging configuration section of an `appsettings.json` or `appsettings.{Environment}.json` file.
- When running behind IIS with the ASP.NET Core Module, `CreateDefaultBuilder` enables IIS Integration, which configures the app's base address and port. IIS Integration also configures the app to capture startup errors. For the IIS default options, see Host ASP.NET Core on Windows with IIS.
- Sets ServiceProviderOptions.ValidateScopes to `true` if the app's environment is Development. For more information, see Scope validation.

The configuration defined by `CreateDefaultBuilder` can be overridden and augmented by ConfigureAppConfiguration, ConfigureLogging, and other methods and extension methods of IWebHostBuilder. A few examples follow:

- ConfigureAppConfiguration is used to specify additional `IConfiguration` for the app. The following `ConfigureAppConfiguration` call adds a delegate to include app configuration in the `appsettings.xml` file. `ConfigureAppConfiguration` may be

called multiple times. Note that this configuration doesn't apply to the host (for example, server URLs or environment). See the Host configuration values section.

```
C#

WebHost.CreateDefaultBuilder(args)
    .ConfigureAppConfiguration((hostingContext, config) =>
    {
        config.AddXmlFile("appsettings.xml", optional: true,
reloadOnChange: true);
    })
    ...
```

- The following `ConfigureLogging` call adds a delegate to configure the minimum logging level (SetMinimumLevel) to LogLevel.Warning. This setting overrides the settings in `appsettings.Development.json` (`LogLevel.Debug`) and `appsettings.Production.json` (`LogLevel.Error`) configured by `CreateDefaultBuilder`. `ConfigureLogging` may be called multiple times.

```
C#

WebHost.CreateDefaultBuilder(args)
    .ConfigureLogging(logging =>
    {
        logging.SetMinimumLevel(LogLevel.Warning);
    })
    ...
```

- The following call to `ConfigureKestrel` overrides the default Limits.MaxRequestBodySize of 30,000,000 bytes established when Kestrel was configured by `CreateDefaultBuilder`:

```
C#

WebHost.CreateDefaultBuilder(args)
    .ConfigureKestrel((context, options) =>
    {
        options.Limits.MaxRequestBodySize = 20000000;
    });
```

The content root determines where the host searches for content files, such as MVC view files. When the app is started from the project's root folder, the project's root folder is used as the content root. This is the default used in Visual Studio and the dotnet new templates.

For more information on app configuration, see Configuration in ASP.NET Core.

> **Note**
>
> As an alternative to using the static `CreateDefaultBuilder` method, creating a host from WebHostBuilder is a supported approach with ASP.NET Core 2.x.

When setting up a host, Configure and ConfigureServices methods can be provided. If a `Startup` class is specified, it must define a `Configure` method. For more information, see App startup in ASP.NET Core. Multiple calls to `ConfigureServices` append to one another. Multiple calls to `Configure` or `UseStartup` on the `WebHostBuilder` replace previous settings.

Host configuration values

WebHostBuilder relies on the following approaches to set the host configuration values:

- Host builder configuration, which includes environment variables with the format `ASPNETCORE_{configurationKey}`. For example, `ASPNETCORE_ENVIRONMENT`.
- Extensions such as UseContentRoot and UseConfiguration (see the Override configuration section).
- UseSetting and the associated key. When setting a value with `UseSetting`, the value is set as a string regardless of the type.

The host uses whichever option sets a value last. For more information, see Override configuration in the next section.

Application Key (Name)

The `IWebHostEnvironment.ApplicationName` property is automatically set when UseStartup or Configure is called during host construction. The value is set to the name of the assembly containing the app's entry point. To set the value explicitly, use the WebHostDefaults.ApplicationKey:

Key: applicationName
Type: *string*
Default: The name of the assembly containing the app's entry point.
Set using: `UseSetting`
Environment variable: `ASPNETCORE_APPLICATIONNAME`

```
C#
```

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.ApplicationKey, "CustomApplicationName")
```

Capture Startup Errors

This setting controls the capture of startup errors.

Key: captureStartupErrors
Type: *bool* (`true` or `1`)
Default: Defaults to `false` unless the app runs with Kestrel behind IIS, where the default is `true`.
Set using: `CaptureStartupErrors`
Environment variable: `ASPNETCORE_CAPTURESTARTUPERRORS`

When `false`, errors during startup result in the host exiting. When `true`, the host captures exceptions during startup and attempts to start the server.

C#

```
WebHost.CreateDefaultBuilder(args)
    .CaptureStartupErrors(true)
```

Content root

This setting determines where ASP.NET Core begins searching for content files.

Key: contentRoot
Type: *string*
Default: Defaults to the folder where the app assembly resides.
Set using: `UseContentRoot`
Environment variable: `ASPNETCORE_CONTENTROOT`

The content root is also used as the base path for the web root. If the content root path doesn't exist, the host fails to start.

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseContentRoot("c:\\<content-root>")
```

For more information, see:

- Fundamentals: Content root
- Web root

Detailed Errors

Determines if detailed errors should be captured.

Key: detailedErrors
Type: *bool* (`true` or `1`)
Default: false
Set using: `UseSetting`
Environment variable: `ASPNETCORE_DETAILEDERRORS`

When enabled (or when the Environment is set to `Development`), the app captures detailed exceptions.

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.DetailedErrorsKey, "true")
```

Environment

Sets the app's environment.

Key: environment
Type: *string*
Default: Production
Set using: `UseEnvironment`
Environment variable: `ASPNETCORE_ENVIRONMENT`

The environment can be set to any value. Framework-defined values include `Development`, `Staging`, and `Production`. Values aren't case sensitive. By default, the *Environment* is read from the `ASPNETCORE_ENVIRONMENT` environment variable. When using Visual Studio, environment variables may be set in the `launchSettings.json` file. For more information, see Use multiple environments in ASP.NET Core.

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseEnvironment(EnvironmentName.Development)
```

Hosting Startup Assemblies

Sets the app's hosting startup assemblies.

Key: hostingStartupAssemblies
Type: *string*
Default: Empty string
Set using: `UseSetting`
Environment variable: `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES`

A semicolon-delimited string of hosting startup assemblies to load on startup.

Although the configuration value defaults to an empty string, the hosting startup assemblies always include the app's assembly. When hosting startup assemblies are provided, they're added to the app's assembly for loading when the app builds its common services during startup.

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.HostingStartupAssembliesKey,
"assembly1;assembly2")
```

HTTPS Port

Set the HTTPS port to redirect to if you get a non-HTTPS connection. Used in enforcing HTTPS. This setting doesn't cause the server to listen on the specified port. That is, it's possible to accidentally redirect requests to an unused port.

Key: https_port
Type: *string*
Default: A default value isn't set.
Set using: `UseSetting`
Environment variable: `ASPNETCORE_HTTPS_PORT`

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting("https_port", "8080")
```

HTTPS Ports

Set the ports to listen on for HTTPS connections.

Key: https_ports **Type**: *string*
Default: A default value isn't set.
Set using: `UseSetting`
Environment variable: `ASPNETCORE_HTTPS_PORTS`

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseSetting("https_ports", "8080")
```

Hosting Startup Exclude Assemblies

A semicolon-delimited string of hosting startup assemblies to exclude on startup.

Key: hostingStartupExcludeAssemblies
Type: *string*
Default: Empty string
Set using: `UseSetting`
Environment variable: `ASPNETCORE_HOSTINGSTARTUPEXCLUDEASSEMBLIES`

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.HostingStartupExcludeAssembliesKey,
"assembly1;assembly2")
```

Prefer Hosting URLs

Indicates whether the host should listen on the URLs configured with the `WebHostBuilder` instead of those configured with the `IServer` implementation.

Key: preferHostingUrls
Type: *bool* (`true` or `1`)
Default: false
Set using: `PreferHostingUrls`
Environment variable: `ASPNETCORE_PREFERHOSTINGURLS`

```
C#

WebHost.CreateDefaultBuilder(args)
```

```
    .PreferHostingUrls(true)
```

Prevent Hosting Startup

Prevents the automatic loading of hosting startup assemblies, including hosting startup assemblies configured by the app's assembly. For more information, see Use hosting startup assemblies in ASP.NET Core.

Key: preventHostingStartup
Type: *bool* (`true` or `1`)
Default: false
Set using: `UseSetting`
Environment variable: `ASPNETCORE_PREVENTHOSTINGSTARTUP`

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.PreventHostingStartupKey, "true")
```

Server URLs

Indicates the IP addresses or host addresses with ports and protocols that the server should listen on for requests.

Key: urls
Type: *string*
Default: http://localhost:5000
Set using: `UseUrls`
Environment variable: `ASPNETCORE_URLS`

Set to a semicolon-separated (;) list of URL prefixes to which the server should respond. For example, `http://localhost:123`. Use "*" to indicate that the server should listen for requests on any IP address or hostname using the specified port and protocol (for example, `http://*:5000`). The protocol (`http://` or `https://`) must be included with each URL. Supported formats vary among servers.

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002")
```

Kestrel has its own endpoint configuration API. For more information, see Configure endpoints for the ASP.NET Core Kestrel web server.

Shutdown Timeout

Specifies the amount of time to wait for Web Host to shut down.

Key: shutdownTimeoutSeconds
Type: *int*
Default: 5
Set using: `UseShutdownTimeout`
Environment variable: `ASPNETCORE_SHUTDOWNTIMEOUTSECONDS`

Although the key accepts an *int* with `UseSetting` (for example, `.UseSetting(WebHostDefaults.ShutdownTimeoutKey, "10")`), the UseShutdownTimeout extension method takes a TimeSpan.

During the timeout period, hosting:

- Triggers IApplicationLifetime.ApplicationStopping.
- Attempts to stop hosted services, logging any errors for services that fail to stop.

If the timeout period expires before all of the hosted services stop, any remaining active services are stopped when the app shuts down. The services stop even if they haven't finished processing. If services require additional time to stop, increase the timeout.

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
```

Startup Assembly

Determines the assembly to search for the `Startup` class.

Key: startupAssembly
Type: *string*
Default: The app's assembly
Set using: `UseStartup`
Environment variable: `ASPNETCORE_STARTUPASSEMBLY`

The assembly by name (`string`) or type (`TStartup`) can be referenced. If multiple `UseStartup` methods are called, the last one takes precedence.

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup("StartupAssemblyName")
```

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<TStartup>()
```

Web root

Sets the relative path to the app's static assets.

Key: webroot
Type: *string*
Default: The default is `wwwroot`. The path to *{content root}/wwwroot* must exist. If the path doesn't exist, a no-op file provider is used.
Set using: `UseWebRoot`
Environment variable: `ASPNETCORE_WEBROOT`

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseWebRoot("public")
```

For more information, see:

- Fundamentals: Web root
- Content root

Override configuration

Use Configuration to configure Web Host. In the following example, host configuration is optionally specified in a `hostsettings.json` file. Any configuration loaded from the `hostsettings.json` file may be overridden by command-line arguments. The built configuration (in `config`) is used to configure the host with UseConfiguration. `IWebHostBuilder` configuration is added to the app's configuration, but the converse isn't true—`ConfigureAppConfiguration` doesn't affect the `IWebHostBuilder` configuration.

Overriding the configuration provided by `UseUrls` with `hostsettings.json` config first, command-line argument config second:

C#

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

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("hostsettings.json", optional: true)
            .AddCommandLine(args)
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseUrls("http://*:5000")
            .UseConfiguration(config)
            .Configure(app =>
            {
                app.Run(context =>
                    context.Response.WriteAsync("Hello, World!"));
            });
    }
}
```

`hostsettings.json`:

JSON

```
{
    urls: "http://*:5005"
}
```

> ⓘ **Note**
>
> UseConfiguration only copies keys from the provided `IConfiguration` to the host builder configuration. Therefore, setting `reloadOnChange: true` for JSON, INI, and XML settings files has no effect.

To specify the host run on a particular URL, the desired value can be passed in from a command prompt when executing dotnet run. The command-line argument overrides

the `urls` value from the `hostsettings.json` file, and the server listens on port 8080:

.NET CLI

```
dotnet run --urls "http://*:8080"
```

Manage the host

Run

The `Run` method starts the web app and blocks the calling thread until the host is shut down:

C#

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

Start

Run the host in a non-blocking manner by calling its `Start` method:

C#

```
using (host)
{
    host.Start();
    Console.ReadLine();
}
```

If a list of URLs is passed to the `Start` method, it listens on the URLs specified:

C#

```
var urls = new List<string>()
{
    "http://*:5000",
    "http://localhost:5001"
};

var host = new WebHostBuilder()
    .UseKestrel()
    .UseStartup<Startup>()
    .Start(urls.ToArray());

using (host)
{
```

```
    Console.ReadLine();
}
```

The app can initialize and start a new host using the pre-configured defaults of `CreateDefaultBuilder` using a static convenience method. These methods start the server without console output and with WaitForShutdown wait for a break (Ctrl-C/SIGINT or SIGTERM):

Start(RequestDelegate app)

Start with a `RequestDelegate`:

C#

```
using (var host = WebHost.Start(app => app.Response.WriteAsync("Hello,
World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Make a request in the browser to `http://localhost:5000` to receive the response "Hello World!" `WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

Start(string url, RequestDelegate app)

Start with a URL and `RequestDelegate`:

C#

```
using (var host = WebHost.Start("http://localhost:8080", app =>
app.Response.WriteAsync("Hello, World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Produces the same result as **Start(RequestDelegate app)**, except the app responds on `http://localhost:8080`.

Start(Action<IRouteBuilder> routeBuilder)

Use an instance of `IRouteBuilder` (Microsoft.AspNetCore.Routing) to use routing middleware:

C#

```
using (var host = WebHost.Start(router => router
    .MapGet("hello/{name}", (req, res, data) =>
        res.WriteAsync($"Hello, {data.Values["name"]}!"))
    .MapGet("buenosdias/{name}", (req, res, data) =>
        res.WriteAsync($"Buenos dias, {data.Values["name"]}!"))
    .MapGet("throw/{message?}", (req, res, data) =>
        throw new Exception((string)data.Values["message"] ?? "Uh oh!"))
    .MapGet("{greeting}/{name}", (req, res, data) =>
        res.WriteAsync($"{data.Values["greeting"]},
{data.Values["name"]}!"))
    .MapGet("", (req, res, data) => res.WriteAsync("Hello, World!"))))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Use the following browser requests with the example:

Expand table

Request	Response
`http://localhost:5000/hello/Martin`	Hello, Martin!
`http://localhost:5000/buenosdias/Catrina`	Buenos dias, Catrina!
`http://localhost:5000/throw/ooops!`	Throws an exception with string "ooops!"
`http://localhost:5000/throw`	Throws an exception with string "Uh oh!"
`http://localhost:5000/Sante/Kevin`	Sante, Kevin!
`http://localhost:5000`	Hello World!

`WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

Start(string url, Action<IRouteBuilder> routeBuilder)

Use a URL and an instance of `IRouteBuilder`:

C#

```
using (var host = WebHost.Start("http://localhost:8080", router => router
    .MapGet("hello/{name}", (req, res, data) =>
        res.WriteAsync($"Hello, {data.Values["name"]}!"))
    .MapGet("buenosdias/{name}", (req, res, data) =>
        res.WriteAsync($"Buenos dias, {data.Values["name"]}!"))
```

```
    .MapGet("throw/{message?}", (req, res, data) =>
        throw new Exception((string)data.Values["message"] ?? "Uh oh!"))
    .MapGet("{greeting}/{name}", (req, res, data) =>
        res.WriteAsync($"{data.Values["greeting"]},
{data.Values["name"]}!"))
    .MapGet("", (req, res, data) => res.WriteAsync("Hello, World!"))))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

Produces the same result as **Start(Action<IRouteBuilder> routeBuilder)**, except the app responds at `http://localhost:8080`.

StartWith(Action<IApplicationBuilder> app)

Provide a delegate to configure an `IApplicationBuilder`:

C#

```
using (var host = WebHost.StartWith(app =>
    app.Use(next =>
    {
        return async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        };
    })))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

Make a request in the browser to `http://localhost:5000` to receive the response "Hello World!" `WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

StartWith(string url, Action<IApplicationBuilder> app)

Provide a URL and a delegate to configure an `IApplicationBuilder`:

C#

```
using (var host = WebHost.StartWith("http://localhost:8080", app =>
    app.Use(next =>
    {
        return async context =>
        {
            await context.Response.WriteAsync("Hello World!");
```

```
        };
    }))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

Produces the same result as **StartWith(Action<IApplicationBuilder> app)**, except the app responds on `http://localhost:8080`.

IWebHostEnvironment interface

The `IWebHostEnvironment` interface provides information about the app's web hosting environment. Use constructor injection to obtain the `IWebHostEnvironment` in order to use its properties and extension methods:

C#

```
public class CustomFileReader
{
    private readonly IWebHostEnvironment _env;

    public CustomFileReader(IWebHostEnvironment env)
    {
        _env = env;
    }

    public string ReadFile(string filePath)
    {
        var fileProvider = _env.WebRootFileProvider;
        // Process the file here
    }
}
```

A convention-based approach can be used to configure the app at startup based on the environment. Alternatively, inject the `IWebHostEnvironment` into the `Startup` constructor for use in `ConfigureServices`:

C#

```
public class Startup
{
    public Startup(IWebHostEnvironment env)
    {
        HostingEnvironment = env;
    }

    public IWebHostEnvironment HostingEnvironment { get; }
```

```
    public void ConfigureServices(IServiceCollection services)
    {
        if (HostingEnvironment.IsDevelopment())
        {
            // Development configuration
        }
        else
        {
            // Staging/Production configuration
        }

        var contentRootPath = HostingEnvironment.ContentRootPath;
    }
}
```

> **Note**
>
> In addition to the `IsDevelopment` extension method, `IWebHostEnvironment` offers `IsStaging`, `IsProduction`, and `IsEnvironment(string environmentName)` methods. For more information, see Use multiple environments in ASP.NET Core.

The `IWebHostEnvironment` service can also be injected directly into the `Configure` method for setting up the processing pipeline:

C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        // In Development, use the Developer Exception Page
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // In Staging/Production, route exceptions to /error
        app.UseExceptionHandler("/error");
    }

    var contentRootPath = env.ContentRootPath;
}
```

`IWebHostEnvironment` can be injected into the `Invoke` method when creating custom middleware:

C#

```
public async Task Invoke(HttpContext context, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        // Configure middleware for Development
    }
    else
    {
        // Configure middleware for Staging/Production
    }

    var contentRootPath = env.ContentRootPath;
}
```

IHostApplicationLifetime interface

`IHostApplicationLifetime` allows for post-startup and shutdown activities. Three properties on the interface are cancellation tokens used to register `Action` methods that define startup and shutdown events.

Expand table

Cancellation Token	Triggered when...
`ApplicationStarted`	The host has fully started.
`ApplicationStopped`	The host is completing a graceful shutdown. All requests should be processed. Shutdown blocks until this event completes.
`ApplicationStopping`	The host is performing a graceful shutdown. Requests may still be processing. Shutdown blocks until this event completes.

C#

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IHostApplicationLifetime
appLifetime)
    {
        appLifetime.ApplicationStarted.Register(OnStarted);
        appLifetime.ApplicationStopping.Register(OnStopping);
        appLifetime.ApplicationStopped.Register(OnStopped);

        Console.CancelKeyPress += (sender, eventArgs) =>
        {
            appLifetime.StopApplication();
            // Don't terminate the process immediately, wait for the Main
thread to exit gracefully.
```

```
            eventArgs.Cancel = true;
        };
    }

    private void OnStarted()
    {
        // Perform post-startup activities here
    }

    private void OnStopping()
    {
        // Perform on-stopping activities here
    }

    private void OnStopped()
    {
        // Perform post-stopped activities here
    }
}
```

`StopApplication` requests termination of the app. The following class uses `StopApplication` to gracefully shut down an app when the class's `Shutdown` method is called:

C#

```
public class MyClass
{
    private readonly IHostApplicationLifetime _appLifetime;

    public MyClass(IHostApplicationLifetime appLifetime)
    {
        _appLifetime = appLifetime;
    }

    public void Shutdown()
    {
        _appLifetime.StopApplication();
    }
}
```

Scope validation

CreateDefaultBuilder sets ServiceProviderOptions.ValidateScopes to `true` if the app's environment is Development.

When `ValidateScopes` is set to `true`, the default service provider performs checks to verify that:

- Scoped services aren't directly or indirectly resolved from the root service provider.
- Scoped services aren't directly or indirectly injected into singletons.

The root service provider is created when BuildServiceProvider is called. The root service provider's lifetime corresponds to the app/server's lifetime when the provider starts with the app and is disposed when the app shuts down.

Scoped services are disposed by the container that created them. If a scoped service is created in the root container, the service's lifetime is effectively promoted to singleton because it's only disposed by the root container when app/server is shut down. Validating service scopes catches these situations when `BuildServiceProvider` is called.

To always validate scopes, including in the Production environment, configure the ServiceProviderOptions with UseDefaultServiceProvider on the host builder:

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseDefaultServiceProvider((context, options) => {
        options.ValidateScopes = true;
    })
```

Additional resources

- Host ASP.NET Core on Windows with IIS
- Host ASP.NET Core on Linux with Nginx
- Host ASP.NET Core in a Windows Service

Configuration in ASP.NET Core

Article • 10/30/2024

By Rick Anderson and Kirk Larkin

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

Application configuration in ASP.NET Core is performed using one or more configuration providers. Configuration providers read configuration data from key-value pairs using a variety of configuration sources:

- Settings files, such as `appsettings.json`
- Environment variables
- Azure Key Vault
- Azure App Configuration
- Command-line arguments
- Custom providers, installed or created
- Directory files
- In-memory .NET objects

This article provides information on configuration in ASP.NET Core. For information on using configuration in console apps, see .NET Configuration.

For Blazor configuration guidance, which adds to or supersedes the guidance in this node, see ASP.NET Core Blazor configuration.

Application and Host Configuration

ASP.NET Core apps configure and launch a *host*. The host is responsible for app startup and lifetime management. The ASP.NET Core templates create a WebApplicationBuilder which contains the host. While some configuration can be done in both the host and the application configuration providers, generally, only configuration that is necessary for the host should be done in host configuration.

Application configuration is the highest priority and is detailed in the next section. Host configuration follows application configuration, and is described in this article.

Default application configuration sources

ASP.NET Core web apps created with dotnet new or Visual Studio generate the following code:

```
C#

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

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults. The initialized `WebApplicationBuilder` (`builder`) provides default configuration for the app in the following order, from highest to lowest priority:

1. Command-line arguments using the Command-line configuration provider.
2. Non-prefixed environment variables using the Non-prefixed environment variables configuration provider.
3. User secrets when the app runs in the `Development` environment.
4. `appsettings.{Environment}.json` using the JSON configuration provider. For example, `appsettings.Production.json` and `appsettings.Development.json`.
5. appsettings.json using the JSON configuration provider.
6. A fallback to the host configuration described in the next section.

Default host configuration sources

The following list contains the default host configuration sources from highest to lowest priority for WebApplicationBuilder:

1. Command-line arguments using the Command-line configuration provider
2. `DOTNET_`-prefixed environment variables using the Environment variables configuration provider.
3. `ASPNETCORE_`-prefixed environment variables using the Environment variables configuration provider.

For the .NET Generic Host and Web Host, the default host configuration sources from highest to lowest priority is:

1. `ASPNETCORE_`-prefixed environment variables using the Environment variables configuration provider.

2. Command-line arguments using the Command-line configuration provider
3. `DOTNET_`-prefixed environment variables using the Environment variables configuration provider.

When a configuration value is set in host and application configuration, the application configuration is used.

Host variables

The following variables are locked in early when initializing the host builders and can't be influenced by application config:

- Application name
- Environment name, for example `Development`, `Production`, and `Staging`
- Content root
- Web root
- Whether to scan for hosting startup assemblies and which assemblies to scan for.
- Variables read by app and library code from HostBuilderContext.Configuration in IHostBuilder.ConfigureAppConfiguration callbacks.

Every other host setting is read from application config instead of host config.

`URLS` is one of the many common host settings that is not a bootstrap setting. Like every other host setting not in the previous list, `URLS` is read later from application config. Host config is a fallback for application config, so host config can be used to set `URLS`, but it will be overridden by any configuration source in application config like `appsettings.json`.

For more information, see Change the content root, app name, and environment and Change the content root, app name, and environment by environment variables or command line

The remaining sections in this article refer to application configuration.

Application configuration providers

The following code displays the enabled configuration providers in the order they were added:

```
C#

public class Index2Model : PageModel
{
```

```
    private IConfigurationRoot ConfigRoot;

    public Index2Model(IConfiguration configRoot)
    {
        ConfigRoot = (IConfigurationRoot)configRoot;
    }

    public ContentResult OnGet()
    {
        string str = "";
        foreach (var provider in ConfigRoot.Providers.ToList())
        {
            str += provider.ToString() + "\n";
        }

        return Content(str);
    }
}
```

The preceding list of highest to lowest priority default configuration sources shows the providers in the opposite order they are added to template generated application. For example, the JSON configuration provider is added before the Command-line configuration provider.

Configuration providers that are added later have higher priority and override previous key settings. For example, if `MyKey` is set in both `appsettings.json` and the environment, the environment value is used. Using the default configuration providers, the Command-line configuration provider overrides all other providers.

For more information on `CreateBuilder`, see Default builder settings.

`appsettings.json`

Consider the following `appsettings.json` file:

JSON

```
{
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  },
  "MyKey": "My appsettings.json Value",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
```

```
  },
  "AllowedHosts": "*"
}
```

The following code from the sample download displays several of the preceding configurations settings:

C#

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

The default JsonConfigurationProvider loads configuration in the following order:

1. `appsettings.json`
2. `appsettings.{Environment}.json` : For example, the `appsettings.Production.json` and `appsettings.Development.json` files. The environment version of the file is loaded based on the IHostingEnvironment.EnvironmentName. For more information, see Use multiple environments in ASP.NET Core.

`appsettings.{Environment}.json` values override keys in `appsettings.json`. For example, by default:

- In development, `appsettings.Development.json` configuration overwrites values found in `appsettings.json`.

- In production, `appsettings.Production.json` configuration overwrites values found in `appsettings.json`. For example, when deploying the app to Azure.

If a configuration value must be guaranteed, see GetValue. The preceding example only reads strings and doesn't support a default value.

Using the default configuration, the `appsettings.json` and `appsettings.{Environment}.json` files are enabled with reloadOnChange: true. Changes made to the `appsettings.json` and `appsettings.{Environment}.json` file ***after*** the app starts are read by the JSON configuration provider.

Comments in appsettings.json

Comments in `appsettings.json` and `appsettings.{Environment}.json` files are supported using JavaScript or C# style comments.

Bind hierarchical configuration data using the options pattern

The preferred way to read related configuration values is using the options pattern. For example, to read the following configuration values:

JSON

```
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  }
```

Create the following `PositionOptions` class:

C#

```
public class PositionOptions
{
    public const string Position = "Position";

    public string Title { get; set; } = String.Empty;
    public string Name { get; set; } = String.Empty;
}
```

An options class:

- Must be non-abstract with a public parameterless constructor.

- All public read-write properties of the type are bound.
- Fields are ***not*** bound. In the preceding code, `Position` is not bound. The `Position` field is used so the string `"Position"` doesn't need to be hard coded in the app when binding the class to a configuration provider.

The following code:

- Calls ConfigurationBinder.Bind to bind the `PositionOptions` class to the `Position` section.
- Displays the `Position` configuration data.

C#

```
public class Test22Model : PageModel
{
    private readonly IConfiguration Configuration;

    public Test22Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var positionOptions = new PositionOptions();

Configuration.GetSection(PositionOptions.Position).Bind(positionOptions);

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

ConfigurationBinder.Get<T> binds and returns the specified type. `ConfigurationBinder.Get<T>` may be more convenient than using `ConfigurationBinder.Bind`. The following code shows how to use `ConfigurationBinder.Get<T>` with the `PositionOptions` class:

C#

```
public class Test21Model : PageModel
{
    private readonly IConfiguration Configuration;
    public PositionOptions? positionOptions { get; private set; }
```

```
    public Test21Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        positionOptions = Configuration.GetSection(PositionOptions.Position)
                                                        .Get<PositionOptions>
();

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

An alternative approach when using the ***options pattern*** is to bind the `Position` section and add it to the dependency injection service container. In the following code, `PositionOptions` is added to the service container with Configure and bound to configuration:

C#

```
using ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<PositionOptions>(
    builder.Configuration.GetSection(PositionOptions.Position));

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

Using the preceding code, the following code reads the position options:

C#

```
public class Test2Model : PageModel
{
    private readonly PositionOptions _options;

    public Test2Model(IOptions<PositionOptions> options)
    {
        _options = options.Value;
    }
```

```
    public ContentResult OnGet()
    {
        return Content($"Title: {_options.Title} \n" +
                       $"Name: {_options.Name}");
    }
}
```

In the preceding code, changes to the JSON configuration file after the app has started are ***not*** read. To read changes after the app has started, use IOptionsSnapshot.

Using the default configuration, the `appsettings.json` and `appsettings.{Environment}.json` files are enabled with reloadOnChange: true. Changes made to the `appsettings.json` and `appsettings.{Environment}.json` file ***after*** the app starts are read by the JSON configuration provider.

See JSON configuration provider in this document for information on adding additional JSON configuration files.

Combining service collection

Consider the following which registers services and configures options:

C#

```
using ConfigSample.Options;
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<PositionOptions>(
    builder.Configuration.GetSection(PositionOptions.Position));
builder.Services.Configure<ColorOptions>(
    builder.Configuration.GetSection(ColorOptions.Color));

builder.Services.AddScoped<IMyDependency, MyDependency>();
builder.Services.AddScoped<IMyDependency2, MyDependency2>();

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

Related groups of registrations can be moved to an extension method to register services. For example, the configuration services are added to the following class:

C#

```
using ConfigSample.Options;
using Microsoft.Extensions.Configuration;

namespace Microsoft.Extensions.DependencyInjection
{
    public static class MyConfigServiceCollectionExtensions
    {
        public static IServiceCollection AddConfig(
             this IServiceCollection services, IConfiguration config)
        {
            services.Configure<PositionOptions>(
                config.GetSection(PositionOptions.Position));
            services.Configure<ColorOptions>(
                config.GetSection(ColorOptions.Color));

            return services;
        }

        public static IServiceCollection AddMyDependencyGroup(
             this IServiceCollection services)
        {
            services.AddScoped<IMyDependency, MyDependency>();
            services.AddScoped<IMyDependency2, MyDependency2>();

            return services;
        }
    }
}
```

The remaining services are registered in a similar class. The following code uses the new extension methods to register the services:

C#

```
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddConfig(builder.Configuration)
    .AddMyDependencyGroup();

builder.Services.AddRazorPages();

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

Note: Each `services.Add{GROUP_NAME}` extension method adds and potentially configures services. For example, AddControllersWithViews adds the services MVC controllers with views require, and AddRazorPages adds the services Razor Pages requires.

Security and user secrets

Configuration data guidelines:

- Never store passwords or other sensitive data in configuration provider code or in plain text configuration files. The Secret Manager tool can be used to store secrets in development.
- Don't use production secrets in development or test environments.
- Specify secrets outside of the project so that they can't be accidentally committed to a source code repository.
- Production apps should use the most secure authentication flow available. For more information, see Secure authentication flows.

By default, the user secrets configuration source is registered after the JSON configuration sources. Therefore, user secrets keys take precedence over keys in `appsettings.json` and `appsettings.{Environment}.json`.

For more information on storing passwords or other sensitive data:

- Use multiple environments in ASP.NET Core
- Safe storage of app secrets in development in ASP.NET Core: Includes advice on using environment variables to store sensitive data. The Secret Manager tool uses the File configuration provider to store user secrets in a JSON file on the local system.
- Azure Key Vault safely stores app secrets for ASP.NET Core apps. For more information, see Azure Key Vault configuration provider in ASP.NET Core.

Non-prefixed environment variables

Non-prefixed environment variables are environment variables other than those prefixed by `ASPNETCORE_` or `DOTNET_`. For example, the ASP.NET Core web application templates set `"ASPNETCORE_ENVIRONMENT": "Development"` in `launchSettings.json`. For more information on `ASPNETCORE_` and `DOTNET_` environment variables, see:

- List of highest to lowest priority default configuration sources including non-prefixed, `ASPNETCORE_`-prefixed and `DOTNETCORE_`-prefixed environment variables.
- DOTNET_ environment variables used outside of Microsoft.Extensions.Hosting.

Using the default configuration, the EnvironmentVariablesConfigurationProvider loads configuration from environment variable key-value pairs after reading `appsettings.json`, `appsettings.{Environment}.json`, and user secrets. Therefore, key

values read from the environment override values read from `appsettings.json`, `appsettings.{Environment}.json`, and user secrets.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` separator is not supported by Bash. The double underscore, `__`, is:

- Supported by all platforms.
- Automatically replaced by a colon, `:`.

The following commands:

- Set the environment keys and values of the preceding example on Windows.
- Test the settings when using the sample download. The `dotnet run` command must be run in the project directory.

.NET CLI

```
set MyKey="My key from Environment"
set Position__Title=Environment_Editor
set Position__Name=Environment_Rick
dotnet run
```

The preceding environment settings:

- Are only set in processes launched from the command window they were set in.
- Won't be read by browsers launched with Visual Studio.

The following setx commands can be used to set the environment keys and values on Windows. Unlike `set`, `setx` settings are persisted. `/M` sets the variable in the system environment. If the `/M` switch isn't used, a user environment variable is set.

Console

```
setx MyKey "My key from setx Environment" /M
setx Position__Title Environment_Editor /M
setx Position__Name Environment_Rick /M
```

To test that the preceding commands override `appsettings.json` and `appsettings.{Environment}.json`:

- With Visual Studio: Exit and restart Visual Studio.
- With the CLI: Start a new command window and enter `dotnet run`.

Call AddEnvironmentVariables with a string to specify a prefix for environment variables:

C#

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

builder.Services.AddRazorPages();

builder.Configuration.AddEnvironmentVariables(prefix: "MyCustomPrefix_");

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

In the preceding code:

- `builder.Configuration.AddEnvironmentVariables(prefix: "MyCustomPrefix_")` is added after the default configuration providers. For an example of ordering the configuration providers, see JSON configuration provider.
- Environment variables set with the `MyCustomPrefix_` prefix override the default configuration providers. This includes environment variables without the prefix.

The prefix is stripped off when the configuration key-value pairs are read.

The following commands test the custom prefix:

.NET CLI

```
set MyCustomPrefix_MyKey="My key with MyCustomPrefix_ Environment"
set MyCustomPrefix_Position__Title=Editor_with_customPrefix
set MyCustomPrefix_Position__Name=Environment_Rick_cp
dotnet run
```

The default configuration loads environment variables and command line arguments prefixed with `DOTNET_` and `ASPNETCORE_`. The `DOTNET_` and `ASPNETCORE_` prefixes are used by ASP.NET Core for host and app configuration, but not for user configuration. For more information on host and app configuration, see .NET Generic Host.

On Azure App Service, select **New application setting** on the **Settings** > **Configuration** page. Azure App Service application settings are:

- Encrypted at rest and transmitted over an encrypted channel.
- Exposed as environment variables.

For more information, see Azure Apps: Override app configuration using the Azure Portal.

See Connection string prefixes for information on Azure database connection strings.

Naming of environment variables

Environment variable names reflect the structure of an `appsettings.json` file. Each element in the hierarchy is separated by a double underscore (preferable) or a colon. When the element structure includes an array, the array index should be treated as an additional element name in this path. Consider the following `appsettings.json` file and its equivalent values represented as environment variables.

`appsettings.json`

JSON

```
{
    "SmtpServer": "smtp.example.com",
    "Logging": [
        {
            "Name": "ToEmail",
            "Level": "Critical",
            "Args": {
                "FromAddress": "MySystem@example.com",
                "ToAddress": "SRE@example.com"
            }
        },
        {
            "Name": "ToConsole",
            "Level": "Information"
        }
    ]
}
```

environment variables

Console

```
setx SmtpServer smtp.example.com
setx Logging__0__Name ToEmail
setx Logging__0__Level Critical
setx Logging__0__Args__FromAddress MySystem@example.com
setx Logging__0__Args__ToAddress SRE@example.com
setx Logging__1__Name ToConsole
setx Logging__1__Level Information
```

Environment variables set in generated launchSettings.json

Environment variables set in `launchSettings.json` override those set in the system environment. For example, the ASP.NET Core web templates generate a `launchSettings.json` file that sets the endpoint configuration to:

JSON

```
"applicationUrl": "https://localhost:5001;http://localhost:5000"
```

Configuring the `applicationUrl` sets the `ASPNETCORE_URLS` environment variable and overrides values set in the environment.

Escape environment variables on Linux

On Linux, the value of URL environment variables must be escaped so `systemd` can parse it. Use the linux tool `systemd-escape` which yields `http:--localhost:5001`

Windows Command Prompt

```
groot@terminus:~$ systemd-escape http://localhost:5001
http:--localhost:5001
```

Display environment variables

The following code displays the environment variables and values on application startup, which can be helpful when debugging environment settings:

C#

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

foreach (var c in builder.Configuration.AsEnumerable())
{
    Console.WriteLine(c.Key + " = " + c.Value);
}
```

Command-line

Using the default configuration, the CommandLineConfigurationProvider loads configuration from command-line argument key-value pairs after the following configuration sources:

- `appsettings.json` and `appsettings.{Environment}.json` files.
- App secrets in the Development environment.
- Environment variables.

By default, configuration values set on the command-line override configuration values set with all the other configuration providers.

Command-line arguments

The following command sets keys and values using `=`:

.NET CLI

```
dotnet run MyKey="Using =" Position:Title=Cmd Position:Name=Cmd_Rick
```

The following command sets keys and values using `/`:

.NET CLI

```
dotnet run /MyKey "Using /" /Position:Title=Cmd /Position:Name=Cmd_Rick
```

The following command sets keys and values using `--`:

.NET CLI

```
dotnet run --MyKey "Using --" --Position:Title=Cmd --Position:Name=Cmd_Rick
```

The key value:

- Must follow `=`, or the key must have a prefix of `--` or `/` when the value follows a space.
- Isn't required if `=` is used. For example, `MySetting=`.

Within the same command, don't mix command-line argument key-value pairs that use `=` with key-value pairs that use a space.

Switch mappings

Switch mappings allow **key** name replacement logic. Provide a dictionary of switch replacements to the AddCommandLine method.

When the switch mappings dictionary is used, the dictionary is checked for a key that matches the key provided by a command-line argument. If the command-line key is found in the dictionary, the dictionary value is passed back to set the key-value pair into the app's configuration. A switch mapping is required for any command-line key prefixed with a single dash (`-`).

Switch mappings dictionary key rules:

- Switches must start with `-` or `--`.
- The switch mappings dictionary must not contain duplicate keys.

To use a switch mappings dictionary, pass it into the call to `AddCommandLine`:

C#

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

builder.Services.AddRazorPages();

var switchMappings = new Dictionary<string, string>()
         {
             { "-k1", "key1" },
             { "-k2", "key2" },
             { "--alt3", "key3" },
             { "--alt4", "key4" },
             { "--alt5", "key5" },
             { "--alt6", "key6" },
         };

builder.Configuration.AddCommandLine(args, switchMappings);

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

Run the following command works to test key replacement:

.NET CLI

```
dotnet run -k1 value1 -k2 value2 --alt3=value2 /alt4=value3 --alt5 value5
/alt6 value6
```

The following code shows the key values for the replaced keys:

C#

```
public class Test3Model : PageModel
{
```

```
    private readonly IConfiguration Config;

    public Test3Model(IConfiguration configuration)
    {
        Config = configuration;
    }

    public ContentResult OnGet()
    {
        return Content(
                $"Key1: '{Config["Key1"]}'\n" +
                $"Key2: '{Config["Key2"]}'\n" +
                $"Key3: '{Config["Key3"]}'\n" +
                $"Key4: '{Config["Key4"]}'\n" +
                $"Key5: '{Config["Key5"]}'\n" +
                $"Key6: '{Config["Key6"]}'");
    }
}
```

For apps that use switch mappings, the call to `CreateDefaultBuilder` shouldn't pass arguments. The `CreateDefaultBuilder` method's `AddCommandLine` call doesn't include mapped switches, and there's no way to pass the switch-mapping dictionary to `CreateDefaultBuilder`. The solution isn't to pass the arguments to `CreateDefaultBuilder` but instead to allow the `ConfigurationBuilder` method's `AddCommandLine` method to process both the arguments and the switch-mapping dictionary.

Set environment and command-line arguments with Visual Studio

Environment and command-line arguments can be set in Visual Studio from the launch profiles dialog:

- In Solution Explorer, right click the project and select **Properties**.
- Select the **Debug** > **General** tab and select **Open debug launch profiles UI**.

Hierarchical configuration data

The Configuration API reads hierarchical configuration data by flattening the hierarchical data with the use of a delimiter in the configuration keys.

The sample download contains the following `appsettings.json` file:

```
JSON
```

```
{
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  },
  "MyKey": "My appsettings.json Value",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

The following code from the sample download displays several of the configurations settings:

C#

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

The preferred way to read hierarchical configuration data is using the options pattern. For more information, see Bind hierarchical configuration data in this document.

GetSection and GetChildren methods are available to isolate sections and children of a section in the configuration data. These methods are described later in GetSection, GetChildren, and Exists.

Configuration keys and values

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

Configuration keys:

- Are case-insensitive. For example, `ConnectionString` and `connectionstring` are treated as equivalent keys.
- If a key and value is set in more than one configuration provider, the value from the last provider added is used. For more information, see Default configuration.
- Hierarchical keys
  - Within the Configuration API, a colon separator (`:`) works on all platforms.
  - In environment variables, a colon separator may not work on all platforms. A double underscore, `__`, is supported by all platforms and is automatically converted into a colon `:`.
  - In Azure Key Vault, hierarchical keys use `--` as a separator. The Azure Key Vault configuration provider automatically replaces `--` with a `:` when the secrets are loaded into the app's configuration.
- The ConfigurationBinder supports binding arrays to objects using array indices in configuration keys. Array binding is described in the Bind an array to a class section.

Configuration values:

- Are strings.
- Null values can't be stored in configuration or bound to objects.

Configuration providers

The following table shows the configuration providers available to ASP.NET Core apps.

Expand table

Provider	Provides configuration from
Azure Key Vault configuration provider	Azure Key Vault
Azure App configuration provider	Azure App Configuration
Command-line configuration provider	Command-line parameters
Custom configuration provider	Custom source
Environment Variables configuration provider	Environment variables
File configuration provider	INI, JSON, and XML files
Key-per-file configuration provider	Directory files
Memory configuration provider	In-memory collections
User secrets	File in the user profile directory

Configuration sources are read in the order that their configuration providers are specified. Order configuration providers in code to suit the priorities for the underlying configuration sources that the app requires.

A typical sequence of configuration providers is:

1. `appsettings.json`
2. `appsettings.{Environment}.json`
3. User secrets
4. Environment variables using the Environment Variables configuration provider.
5. Command-line arguments using the Command-line configuration provider.

A common practice is to add the Command-line configuration provider last in a series of providers to allow command-line arguments to override configuration set by the other providers.

The preceding sequence of providers is used in the default configuration.

Connection string prefixes

⚠ **Warning**

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

The Configuration API has special processing rules for four connection string environment variables. These connection strings are involved in configuring Azure connection strings for the app environment. Environment variables with the prefixes shown in the table are loaded into the app with the default configuration or when no prefix is supplied to `AddEnvironmentVariables`.

Expand table

Connection string prefix	Provider
`CUSTOMCONNSTR_`	Custom provider
`MYSQLCONNSTR_`	MySQL
`SQLAZURECONNSTR_`	Azure SQL Database
`SQLCONNSTR_`	SQL Server

When an environment variable is discovered and loaded into configuration with any of the four prefixes shown in the table:

- The configuration key is created by removing the environment variable prefix and adding a configuration key section (`ConnectionStrings`).
- A new configuration key-value pair is created that represents the database connection provider (except for `CUSTOMCONNSTR_`, which has no stated provider).

Expand table

Environment variable key	Converted configuration key	Provider configuration entry
`CUSTOMCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Configuration entry not created.
`MYSQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` Value: `MySql.Data.MySqlClient`

Environment variable key	Converted configuration key	Provider configuration entry
`SQLAZURECONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName`: Value: `System.Data.SqlClient`
`SQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName`: Value: `System.Data.SqlClient`

File configuration provider

FileConfigurationProvider is the base class for loading configuration from the file system. The following configuration providers derive from `FileConfigurationProvider`:

- INI configuration provider
- JSON configuration provider
- XML configuration provider

INI configuration provider

The IniConfigurationProvider loads configuration from INI file key-value pairs at runtime.

The following code adds several configuration providers:

C#

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

builder.Configuration
    .AddIniFile("MyIniConfig.ini", optional: true, reloadOnChange: true)
    .AddIniFile($"MyIniConfig.{builder.Environment.EnvironmentName}.ini",
                optional: true, reloadOnChange: true);

builder.Configuration.AddEnvironmentVariables();
builder.Configuration.AddCommandLine(args);

builder.Services.AddRazorPages();

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

In the preceding code, settings in the `MyIniConfig.ini` and `MyIniConfig.{Environment}.ini` files are overridden by settings in the:

- Environment variables configuration provider

- Command-line configuration provider.

The sample download contains the following `MyIniConfig.ini` file:

ini

```ini
MyKey="MyIniConfig.ini Value"

[Position]
Title="My INI Config title"
Name="My INI Config name"

[Logging:LogLevel]
Default=Information
Microsoft=Warning
```

The following code from the sample download displays several of the preceding configurations settings:

C#

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

JSON configuration provider

The JsonConfigurationProvider loads configuration from JSON file key-value pairs.

Overloads can specify:

- Whether the file is optional.
- Whether the configuration is reloaded if the file changes.

Consider the following code:

```
C#
```

```
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddJsonFile("MyConfig.json",
        optional: true,
        reloadOnChange: true);

builder.Services.AddRazorPages();

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

The preceding code:

- Configures the JSON configuration provider to load the `MyConfig.json` file with the following options:
  - `optional: true`: The file is optional.
  - `reloadOnChange: true` : The file is reloaded when changes are saved.
- Reads the default configuration providers before the `MyConfig.json` file. Settings in the `MyConfig.json` file override setting in the default configuration providers, including the Environment variables configuration provider and the Command-line configuration provider.

You typically ***don't*** want a custom JSON file overriding values set in the Environment variables configuration provider and the Command-line configuration provider.

XML configuration provider

The XmlConfigurationProvider loads configuration from XML file key-value pairs at runtime.

The following code adds several configuration providers:

```
C#
```

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

```
builder.Configuration
    .AddXmlFile("MyXMLFile.xml", optional: true, reloadOnChange: true)
    .AddXmlFile($"MyXMLFile.{builder.Environment.EnvironmentName}.xml",
                optional: true, reloadOnChange: true);

builder.Configuration.AddEnvironmentVariables();
builder.Configuration.AddCommandLine(args);

builder.Services.AddRazorPages();

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

In the preceding code, settings in the `MyXMLFile.xml` and `MyXMLFile.{Environment}.xml` files are overridden by settings in the:

- Environment variables configuration provider
- Command-line configuration provider.

The sample download contains the following `MyXMLFile.xml` file:

XML

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <MyKey>MyXMLFile Value</MyKey>
  <Position>
    <Title>Title from  MyXMLFile</Title>
    <Name>Name from MyXMLFile</Name>
  </Position>
  <Logging>
    <LogLevel>
      <Default>Information</Default>
      <Microsoft>Warning</Microsoft>
    </LogLevel>
  </Logging>
</configuration>
```

The following code from the sample download displays several of the preceding configurations settings:

C#

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
```

```
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

Repeating elements that use the same element name work if the `name` attribute is used to distinguish the elements:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <section name="section0">
    <key name="key0">value 00</key>
    <key name="key1">value 01</key>
  </section>
  <section name="section1">
    <key name="key0">value 10</key>
    <key name="key1">value 11</key>
  </section>
</configuration>
```

The following code reads the previous configuration file and displays the keys and values:

C#

```
public class IndexModel : PageModel
{
    private readonly IConfiguration Configuration;

    public IndexModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var key00 = "section:section0:key:key0";
```

```
        var key01 = "section:section0:key:key1";
        var key10 = "section:section1:key:key0";
        var key11 = "section:section1:key:key1";

        var val00 = Configuration[key00];
        var val01 = Configuration[key01];
        var val10 = Configuration[key10];
        var val11 = Configuration[key11];

        return Content($"{key00} value: {val00} \n" +
                       $"{key01} value: {val01} \n" +
                       $"{key10} value: {val10} \n" +
                       $"{key10} value: {val11} \n"
                       );
    }
}
```

Attributes can be used to supply values:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <key attribute="value" />
  <section>
    <key attribute="value" />
  </section>
</configuration>
```

The previous configuration file loads the following keys with `value`:

- key:attribute
- section:key:attribute

Key-per-file configuration provider

The KeyPerFileConfigurationProvider uses a directory's files as configuration key-value pairs. The key is the file name. The value contains the file's contents. The Key-per-file configuration provider is used in Docker hosting scenarios.

To activate key-per-file configuration, call the AddKeyPerFile extension method on an instance of ConfigurationBuilder. The `directoryPath` to the files must be an absolute path.

Overloads permit specifying:

- An `Action<KeyPerFileConfigurationSource>` delegate that configures the source.

- Whether the directory is optional and the path to the directory.

The double-underscore (`__`) is used as a configuration key delimiter in file names. For example, the file name `Logging__LogLevel__System` produces the configuration key `Logging:LogLevel:System`.

Call `ConfigureAppConfiguration` when building the host to specify the app's configuration:

C#

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    var path = Path.Combine(
        Directory.GetCurrentDirectory(), "path/to/files");
    config.AddKeyPerFile(directoryPath: path, optional: true);
})
```

Memory configuration provider

The MemoryConfigurationProvider uses an in-memory collection as configuration key-value pairs.

The following code adds a memory collection to the configuration system:

C#

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

var Dict = new Dictionary<string, string>
        {
           {"MyKey", "Dictionary MyKey Value"},
           {"Position:Title", "Dictionary_Title"},
           {"Position:Name", "Dictionary_Name" },
           {"Logging:LogLevel:Default", "Warning"}
        };

builder.Configuration.AddInMemoryCollection(Dict);
builder.Configuration.AddEnvironmentVariables();
builder.Configuration.AddCommandLine(args);

builder.Services.AddRazorPages();

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

The following code from the sample download displays the preceding configurations settings:

C#

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

In the preceding code, `config.AddInMemoryCollection(Dict)` is added after the default configuration providers. For an example of ordering the configuration providers, see JSON configuration provider.

See Bind an array for another example using `MemoryConfigurationProvider`.

Kestrel endpoint configuration

Kestrel specific endpoint configuration overrides all cross-server endpoint configurations. Cross-server endpoint configurations include:

- UseUrls
- `--urls` on the command line
- The environment variable `ASPNETCORE_URLS`

Consider the following `appsettings.json` file used in an ASP.NET Core web app:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "Https": {
        "Url": "https://localhost:9999"
      }
    }
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

When the preceding highlighted markup is used in an ASP.NET Core web app ***and*** the app is launched on the command line with the following cross-server endpoint configuration:

```
dotnet run --urls="https://localhost:7777"
```

Kestrel binds to the endpoint configured specifically for Kestrel in the `appsettings.json` file (`https://localhost:9999`) and not `https://localhost:7777`.

Consider the Kestrel specific endpoint configured as an environment variable:

```
set Kestrel__Endpoints__Https__Url=https://localhost:8888
```

In the preceding environment variable, `Https` is the name of the Kestrel specific endpoint. The preceding `appsettings.json` file also defines a Kestrel specific endpoint named `Https`. By default, environment variables using the Environment Variables configuration provider are read after `appsettings.{Environment}.json`, therefore, the preceding environment variable is used for the `Https` endpoint.

GetValue

ConfigurationBinder.GetValue extracts a single value from configuration with a specified key and converts it to the specified type:

```
C#

public class TestNumModel : PageModel
{
```

```
    private readonly IConfiguration Configuration;

    public TestNumModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var number = Configuration.GetValue<int>("NumberKey", 99);
        return Content($"{number}");
    }
}
```

In the preceding code, if `NumberKey` isn't found in the configuration, the default value of `99` is used.

GetSection, GetChildren, and Exists

For the examples that follow, consider the following `MySubsection.json` file:

JSON

```
{
  "section0": {
    "key0": "value00",
    "key1": "value01"
  },
  "section1": {
    "key0": "value10",
    "key1": "value11"
  },
  "section2": {
    "subsection0": {
      "key0": "value200",
      "key1": "value201"
    },
    "subsection1": {
      "key0": "value210",
      "key1": "value211"
    }
  }
}
```

The following code adds `MySubsection.json` to the configuration providers:

C#

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

builder.Configuration
    .AddJsonFile("MySubsection.json",
                 optional: true,
                 reloadOnChange: true);

builder.Services.AddRazorPages();

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

GetSection

IConfiguration.GetSection returns a configuration subsection with the specified subsection key.

The following code returns values for `section1`:

C#

```
public class TestSectionModel : PageModel
{
    private readonly IConfiguration Config;

    public TestSectionModel(IConfiguration configuration)
    {
        Config = configuration.GetSection("section1");
    }

    public ContentResult OnGet()
    {
        return Content(
                $"section1:key0: '{Config["key0"]}'\n" +
                $"section1:key1: '{Config["key1"]}'");
    }
}
```

The following code returns values for `section2:subsection0`:

C#

```
public class TestSection2Model : PageModel
{
    private readonly IConfiguration Config;

    public TestSection2Model(IConfiguration configuration)
    {
        Config = configuration.GetSection("section2:subsection0");
    }
```

```
    public ContentResult OnGet()
    {
        return Content(
                $"section2:subsection0:key0 '{Config["key0"]}'\n" +
                $"section2:subsection0:key1:'{Config["key1"]}'");
    }
}
```

`GetSection` never returns `null`. If a matching section isn't found, an empty `IConfigurationSection` is returned.

When `GetSection` returns a matching section, Value isn't populated. A Key and Path are returned when the section exists.

GetChildren and Exists

The following code calls IConfiguration.GetChildren and returns values for `section2:subsection0`:

C#

```
public class TestSection4Model : PageModel
{
    private readonly IConfiguration Config;

    public TestSection4Model(IConfiguration configuration)
    {
        Config = configuration;
    }

    public ContentResult OnGet()
    {
        string s = "";
        var selection = Config.GetSection("section2");
        if (!selection.Exists())
        {
            throw new Exception("section2 does not exist.");
        }
        var children = selection.GetChildren();

        foreach (var subSection in children)
        {
            int i = 0;
            var key1 = subSection.Key + ":key" + i++.ToString();
            var key2 = subSection.Key + ":key" + i.ToString();
            s += key1 + " value: " + selection[key1] + "\n";
            s += key2 + " value: " + selection[key2] + "\n";
        }
        return Content(s);
```

```
    }
}
```

The preceding code calls ConfigurationExtensions.Exists to verify the section exists:

Bind an array

The ConfigurationBinder.Bind supports binding arrays to objects using array indices in configuration keys. Any array format that exposes a numeric key segment is capable of array binding to a POCO class array.

Consider `MyArray.json` from the sample download:

JSON

```
{
  "array": {
    "entries": {
      "0": "value00",
      "1": "value10",
      "2": "value20",
      "4": "value40",
      "5": "value50"
    }
  }
}
```

The following code adds `MyArray.json` to the configuration providers:

C#

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

builder.Configuration
    .AddJsonFile("MyArray.json",
                 optional: true,
                 reloadOnChange: true);

builder.Services.AddRazorPages();

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

The following code reads the configuration and displays the values:

C#

```
public class ArrayModel : PageModel
{
    private readonly IConfiguration Config;
    public ArrayExample? _array { get; private set; }

    public ArrayModel(IConfiguration config)
    {
        Config = config;
    }

    public ContentResult OnGet()
    {
       _array = Config.GetSection("array").Get<ArrayExample>();
        if (_array == null)
        {
            throw new ArgumentNullException(nameof(_array));
        }
        string s = String.Empty;

        for (int j = 0; j < _array.Entries.Length; j++)
        {
            s += $"Index: {j}  Value:  {_array.Entries[j]} \n";
        }

        return Content(s);
    }
}
```

C#

```
public class ArrayExample
{
    public string[]? Entries { get; set; }
}
```

The preceding code returns the following output:

text

```
Index: 0  Value: value00
Index: 1  Value: value10
Index: 2  Value: value20
Index: 3  Value: value40
Index: 4  Value: value50
```

In the preceding output, Index 3 has value `value40`, corresponding to `"4": "value40",` in `MyArray.json`. The bound array indices are continuous and not bound to the configuration key index. The configuration binder isn't capable of binding null values or creating null entries in bound objects.

Custom configuration provider

The sample app demonstrates how to create a basic configuration provider that reads configuration key-value pairs from a database using Entity Framework (EF).

The provider has the following characteristics:

- The EF in-memory database is used for demonstration purposes. To use a database that requires a connection string, implement a secondary `ConfigurationBuilder` to supply the connection string from another configuration provider.
- The provider reads a database table into configuration at startup. The provider doesn't query the database on a per-key basis.
- Reload-on-change isn't implemented, so updating the database after the app starts has no effect on the app's configuration.

Define an `EFConfigurationValue` entity for storing configuration values in the database.

`Models/EFConfigurationValue.cs`:

C#

```
public class EFConfigurationValue
{
    public string Id { get; set; } = String.Empty;
    public string Value { get; set; } = String.Empty;
}
```

Add an `EFConfigurationContext` to store and access the configured values.

`EFConfigurationProvider/EFConfigurationContext.cs`:

C#

```
public class EFConfigurationContext : DbContext
{
    public EFConfigurationContext(DbContextOptions<EFConfigurationContext>
options) : base(options)
    {
    }

    public DbSet<EFConfigurationValue> Values => Set<EFConfigurationValue>
();
}
```

Create a class that implements IConfigurationSource.

`EFConfigurationProvider/EFConfigurationSource.cs`:

```
C#

public class EFConfigurationSource : IConfigurationSource
{
    private readonly Action<DbContextOptionsBuilder> _optionsAction;

    public EFConfigurationSource(Action<DbContextOptionsBuilder>
optionsAction) => _optionsAction = optionsAction;

    public IConfigurationProvider Build(IConfigurationBuilder builder) =>
new EFConfigurationProvider(_optionsAction);
}
```

Create the custom configuration provider by inheriting from ConfigurationProvider. The configuration provider initializes the database when it's empty. Since configuration keys are case-insensitive, the dictionary used to initialize the database is created with the case-insensitive comparer (StringComparer.OrdinalIgnoreCase).

`EFConfigurationProvider/EFConfigurationProvider.cs`:

```
C#

public class EFConfigurationProvider : ConfigurationProvider
{
    public EFConfigurationProvider(Action<DbContextOptionsBuilder>
optionsAction)
    {
        OptionsAction = optionsAction;
    }

    Action<DbContextOptionsBuilder> OptionsAction { get; }

    public override void Load()
    {
        var builder = new DbContextOptionsBuilder<EFConfigurationContext>();

        OptionsAction(builder);

        using (var dbContext = new EFConfigurationContext(builder.Options))
        {
            if (dbContext == null || dbContext.Values == null)
            {
                throw new Exception("Null DB context");
            }
            dbContext.Database.EnsureCreated();

            Data = !dbContext.Values.Any()
                ? CreateAndSaveDefaultValues(dbContext)
                : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
```

```
        }
    }

    private static IDictionary<string, string> CreateAndSaveDefaultValues(
        EFConfigurationContext dbContext)
    {
        // Quotes (c)2005 Universal Pictures: Serenity
        // https://www.uphe.com/movies/serenity-2005
        var configValues =
            new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase)
            {
                    { "quote1", "I aim to misbehave." },
                    { "quote2", "I swallowed a bug." },
                    { "quote3", "You can't stop the signal, Mal." }
            };

        if (dbContext == null || dbContext.Values == null)
        {
            throw new Exception("Null DB context");
        }

        dbContext.Values.AddRange(configValues
            .Select(kvp => new EFConfigurationValue
            {
                Id = kvp.Key,
                Value = kvp.Value
            })
            .ToArray());

        dbContext.SaveChanges();

        return configValues;
    }
}
```

An `AddEFConfiguration` extension method permits adding the configuration source to a `ConfigurationBuilder`.

`Extensions/EntityFrameworkExtensions.cs`:

C#

```
public static class EntityFrameworkExtensions
{
    public static IConfigurationBuilder AddEFConfiguration(
                this IConfigurationBuilder builder,
                Action<DbContextOptionsBuilder> optionsAction)
    {
        return builder.Add(new EFConfigurationSource(optionsAction));
    }
}
```

The following code shows how to use the custom `EFConfigurationProvider` in `Program.cs`:

C#

```
//using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddEFConfiguration(
    opt => opt.UseInMemoryDatabase("InMemoryDb"));

var app = builder.Build();

app.Run();
```

Access configuration with Dependency Injection (DI)

Configuration can be injected into services using Dependency Injection (DI) by resolving the IConfiguration service:

C#

```
public class Service
{
    private readonly IConfiguration _config;

    public Service(IConfiguration config) =>
        _config = config;

    public void DoSomething()
    {
        var configSettingValue = _config["ConfigSetting"];

        // ...
    }
}
```

For information on how to access values using `IConfiguration`, see GetValue and GetSection, GetChildren, and Exists in this article.

Access configuration in Razor Pages

The following code displays configuration data in a Razor Page:

CSHTML

```
@page
@model Test5Model
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

Configuration value for 'MyKey': @Configuration["MyKey"]
```

In the following code, `MyOptions` is added to the service container with Configure and bound to configuration:

C#

```
using SampleApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<MyOptions>(
    builder.Configuration.GetSection("MyOptions"));

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

The following markup uses the @inject Razor directive to resolve and display the options values:

CSHTML

```
@page
@model SampleApp.Pages.Test3Model
@using Microsoft.Extensions.Options
@using SampleApp.Models
@inject IOptions<MyOptions> optionsAccessor


<p><b>Option1:</b> @optionsAccessor.Value.Option1</p>
<p><b>Option2:</b> @optionsAccessor.Value.Option2</p>
```

Access configuration in a MVC view file

The following code displays configuration data in a MVC view:

CSHTML

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

Configuration value for 'MyKey': @Configuration["MyKey"]
```

Access configuration in `Program.cs`

The following code accesses configuration in the `Program.cs` file.

C#

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

var key1 = builder.Configuration.GetValue<string>("KeyOne");

var app = builder.Build();

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

var key2 = app.Configuration.GetValue<int>("KeyTwo");
var key3 = app.Configuration.GetValue<bool>("KeyThree");

app.Logger.LogInformation("KeyOne: {KeyOne}", key1);
app.Logger.LogInformation("KeyTwo: {KeyTwo}", key2);
app.Logger.LogInformation("KeyThree: {KeyThree}", key3);

app.Run();
```

In `appsettings.json` for the preceding example:

JSON

```
{
  ...
  "KeyOne": "Key One Value",
  "KeyTwo": 1999,
  "KeyThree": true
}
```

Configure options with a delegate

Options configured in a delegate override values set in the configuration providers.

In the following code, an IConfigureOptions<TOptions> service is added to the service container. It uses a delegate to configure values for `MyOptions`:

C#

```
using SampleApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<MyOptions>(myOptions =>
{
    myOptions.Option1 = "Value configured in delegate";
    myOptions.Option2 = 500;
});

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

The following code displays the options values:

C#

```
public class Test2Model : PageModel
{
    private readonly IOptions<MyOptions> _optionsDelegate;

    public Test2Model(IOptions<MyOptions> optionsDelegate )
    {
        _optionsDelegate = optionsDelegate;
    }

    public ContentResult OnGet()
    {
        return Content($"Option1: {_optionsDelegate.Value.Option1} \n" +
                       $"Option2: {_optionsDelegate.Value.Option2}");
    }
}
```

In the preceding example, the values of `Option1` and `Option2` are specified in `appsettings.json` and then overridden by the configured delegate.

Host versus app configuration

Before the app is configured and started, a *host* is configured and launched. The host is responsible for app startup and lifetime management. Both the app and the host are configured using the configuration providers described in this topic. Host configuration key-value pairs are also included in the app's configuration. For more information on how the configuration providers are used when the host is built and how configuration sources affect host configuration, see ASP.NET Core fundamentals overview.

Default host configuration

For details on the default configuration when using the Web Host, see the ASP.NET Core 2.2 version of this topic.

- Host configuration is provided from:
  - Environment variables prefixed with `DOTNET_` (for example, `DOTNET_ENVIRONMENT`) using the Environment Variables configuration provider. The prefix (`DOTNET_`) is stripped when the configuration key-value pairs are loaded.
  - Command-line arguments using the Command-line configuration provider.
- Web Host default configuration is established (`ConfigureWebHostDefaults`):
  - Kestrel is used as the web server and configured using the app's configuration providers.
  - Add Host Filtering Middleware.
  - Add Forwarded Headers Middleware if the `ASPNETCORE_FORWARDEDHEADERS_ENABLED` environment variable is set to `true`.
  - Enable IIS integration.

Other configuration

This topic only pertains to *app configuration*. Other aspects of running and hosting ASP.NET Core apps are configured using configuration files not covered in this topic:

- `launch.json`/`launchSettings.json` are tooling configuration files for the Development environment, described:
  - In Use multiple environments in ASP.NET Core.
  - Across the documentation set where the files are used to configure ASP.NET Core apps for Development scenarios.
- `web.config` is a server configuration file, described in the following topics:
  - Host ASP.NET Core on Windows with IIS
  - ASP.NET Core Module (ANCM) for IIS

Environment variables set in `launchSettings.json` override those set in the system environment.

For more information on migrating app configuration from earlier versions of ASP.NET, see Update from ASP.NET to ASP.NET Core.

Add configuration from an external assembly

An IHostingStartup implementation allows adding enhancements to an app at startup from an external assembly outside of the app's `Startup` class. For more information, see Use hosting startup assemblies in ASP.NET Core.

Configuration-binding source generator

The Configuration-binding source generator provides AOT and trim-friendly configuration. For more information, see Configuration-binding source generator.

Additional resources

- Configuration source code
- WebApplicationBuilder source code
- View or download sample code (how to download)
- Options pattern in ASP.NET Core
- ASP.NET Core Blazor configuration

Options pattern in ASP.NET Core

Article • 10/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson.

The options pattern uses classes to provide strongly typed access to groups of related settings. When configuration settings are isolated by scenario into separate classes, the app adheres to two important software engineering principles:

- Encapsulation:
  - Classes that depend on configuration settings depend only on the configuration settings that they use.
- Separation of Concerns:
  - Settings for different parts of the app aren't dependent or coupled to one another.

Options also provide a mechanism to validate configuration data. For more information, see the Options validation section.

This article provides information on the options pattern in ASP.NET Core. For information on using the options pattern in console apps, see Options pattern in .NET.

Bind hierarchical configuration

The preferred way to read related configuration values is using the options pattern. For example, to read the following configuration values:

JSON

```
"Position": {
  "Title": "Editor",
  "Name": "Joe Smith"
}
```

Create the following `PositionOptions` class:

C#

```
public class PositionOptions
{
    public const string Position = "Position";

    public string Title { get; set; } = String.Empty;
    public string Name { get; set; } = String.Empty;
}
```

An options class:

- Must be non-abstract.
- Has public read-write properties of the type that have corresponding items in config are bound.
- Has its read-write properties bound to matching entries in configuration.
- Does ***not*** have its fields bound. In the preceding code, `Position` is not bound. The `Position` field is used so the string `"Position"` doesn't need to be hard coded in the app when binding the class to a configuration provider.

The following code:

- Calls ConfigurationBinder.Bind to bind the `PositionOptions` class to the `Position` section.
- Displays the `Position` configuration data.

C#

```
public class Test22Model : PageModel
{
    private readonly IConfiguration Configuration;

    public Test22Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var positionOptions = new PositionOptions();

Configuration.GetSection(PositionOptions.Position).Bind(positionOptions);

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
```

```
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

ConfigurationBinder.Get<T> binds and returns the specified type. `ConfigurationBinder.Get<T>` may be more convenient than using `ConfigurationBinder.Bind`. The following code shows how to use `ConfigurationBinder.Get<T>` with the `PositionOptions` class:

C#

```
public class Test21Model : PageModel
{
    private readonly IConfiguration Configuration;
    public PositionOptions? positionOptions { get; private set; }

    public Test21Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        positionOptions = Configuration.GetSection(PositionOptions.Position)
                                                    .Get<PositionOptions>
();

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

Bind also allows the concretion of an abstract class. Consider the following code which uses the abstract class `SomethingWithAName`:

C#

```
namespace ConfigSample.Options;

public abstract class SomethingWithAName
{
    public abstract string? Name { get; set; }
}
```

```
public class NameTitleOptions(int age) : SomethingWithAName
{
    public const string NameTitle = "NameTitle";

    public override string? Name { get; set; }
    public string Title { get; set; } = string.Empty;

    public int Age { get; set; } = age;
}
```

The following code displays the `NameTitleOptions` configuration values:

C#

```
public class Test33Model : PageModel
{
    private readonly IConfiguration Configuration;

    public Test33Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var nameTitleOptions = new NameTitleOptions(22);

Configuration.GetSection(NameTitleOptions.NameTitle).Bind(nameTitleOptions);

        return Content($"Title: {nameTitleOptions.Title} \n" +
                       $"Name: {nameTitleOptions.Name}  \n" +
                       $"Age: {nameTitleOptions.Age}"
                       );
    }
}
```

Calls to `Bind` are less strict than calls to `Get<>`:

- `Bind` allows the concretion of an abstract.
- `Get<>` has to create an instance itself.

The Options Pattern

An alternative approach when using the ***options pattern*** is to bind the `Position` section and add it to the dependency injection service container. In the following code, `PositionOptions` is added to the service container with Configure and bound to configuration:

C#

```
using ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<PositionOptions>(
    builder.Configuration.GetSection(PositionOptions.Position));

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

Using the preceding code, the following code reads the position options:

C#

```
public class Test2Model : PageModel
{
    private readonly PositionOptions _options;

    public Test2Model(IOptions<PositionOptions> options)
    {
        _options = options.Value;
    }

    public ContentResult OnGet()
    {
        return Content($"Title: {_options.Title} \n" +
                       $"Name: {_options.Name}");
    }
}
```

In the preceding code, changes to the JSON configuration file after the app has started are ***not*** read. To read changes after the app has started, use IOptionsSnapshot.

Options interfaces

IOptions<TOptions>:

- Does ***not*** support:
  - Reading of configuration data after the app has started.
  - Named options
- Is registered as a Singleton and can be injected into any service lifetime.

IOptionsSnapshot<TOptions>:

- Is useful in scenarios where options should be recomputed on every request. For more information, see Use IOptionsSnapshot to read updated data.
- Is registered as Scoped and therefore can't be injected into a Singleton service.
- Supports named options

IOptionsMonitor<TOptions>:

- Is used to retrieve options and manage options notifications for `TOptions` instances.
- Is registered as a Singleton and can be injected into any service lifetime.
- Supports:
  - Change notifications
  - named options
  - Reloadable configuration
  - Selective options invalidation (IOptionsMonitorCache<TOptions>)

Post-configuration scenarios enable setting or changing options after all IConfigureOptions<TOptions> configuration occurs.

IOptionsFactory<TOptions> is responsible for creating new options instances. It has a single Create method. The default implementation takes all registered IConfigureOptions<TOptions> and IPostConfigureOptions<TOptions> and runs all the configurations first, followed by the post-configuration. It distinguishes between IConfigureNamedOptions<TOptions> and IConfigureOptions<TOptions> and only calls the appropriate interface.

IOptionsMonitorCache<TOptions> is used by IOptionsMonitor<TOptions> to cache `TOptions` instances. The IOptionsMonitorCache<TOptions> invalidates options instances in the monitor so that the value is recomputed (TryRemove). Values can be manually introduced with TryAdd. The Clear method is used when all named instances should be recreated on demand.

Use IOptionsSnapshot to read updated data

Using IOptionsSnapshot<TOptions>:

- Options are computed once per request when accessed and cached for the lifetime of the request.
- May incur a significant performance penalty because it's a Scoped service and is recomputed per request. For more information, see this GitHub issue and Improve the performance of configuration binding.

- Changes to the configuration are read after the app starts when using configuration providers that support reading updated configuration values.

The difference between `IOptionsMonitor` and `IOptionsSnapshot` is that:

- `IOptionsMonitor` is a Singleton service that retrieves current option values at any time, which is especially useful in singleton dependencies.
- `IOptionsSnapshot` is a Scoped service and provides a snapshot of the options at the time the `IOptionsSnapshot<T>` object is constructed. Options snapshots are designed for use with transient and scoped dependencies.

The following code uses IOptionsSnapshot<TOptions>.

C#

```
public class TestSnapModel : PageModel
{
    private readonly MyOptions _snapshotOptions;

    public TestSnapModel(IOptionsSnapshot<MyOptions>
snapshotOptionsAccessor)
    {
        _snapshotOptions = snapshotOptionsAccessor.Value;
    }

    public ContentResult OnGet()
    {
        return Content($"Option1: {_snapshotOptions.Option1} \n" +
                       $"Option2: {_snapshotOptions.Option2}");
    }
}
```

The following code registers a configuration instance which `MyOptions` binds against:

C#

```
using SampleApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<MyOptions>(
    builder.Configuration.GetSection("MyOptions"));

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

In the preceding code, changes to the JSON configuration file after the app has started are read.

IOptionsMonitor

The following code registers a configuration instance which `MyOptions` binds against.

C#

```
using SampleApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<MyOptions>(
    builder.Configuration.GetSection("MyOptions"));

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

The following example uses IOptionsMonitor<TOptions>:

C#

```
public class TestMonitorModel : PageModel
{
    private readonly IOptionsMonitor<MyOptions> _optionsDelegate;

    public TestMonitorModel(IOptionsMonitor<MyOptions> optionsDelegate )
    {
        _optionsDelegate = optionsDelegate;
    }

    public ContentResult OnGet()
    {
        return Content($"Option1: {_optionsDelegate.CurrentValue.Option1}
\n" +
                       $"Option2: {_optionsDelegate.CurrentValue.Option2}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

Named options support using IConfigureNamedOptions

Named options:

- Are useful when multiple configuration sections bind to the same properties.
- Are case sensitive.

Consider the following `appsettings.json` file:

JSON

```
{
  "TopItem": {
    "Month": {
      "Name": "Green Widget",
      "Model": "GW46"
    },
    "Year": {
      "Name": "Orange Gadget",
      "Model": "OG35"
    }
  }
}
```

Rather than creating two classes to bind `TopItem:Month` and `TopItem:Year`, the following class is used for each section:

C#

```
public class TopItemSettings
{
    public const string Month = "Month";
    public const string Year = "Year";

    public string Name { get; set; } = string.Empty;
    public string Model { get; set; } = string.Empty;
}
```

The following code configures the named options:

C#

```
using SampleApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<TopItemSettings>(TopItemSettings.Month,
    builder.Configuration.GetSection("TopItem:Month"));
builder.Services.Configure<TopItemSettings>(TopItemSettings.Year,
    builder.Configuration.GetSection("TopItem:Year"));
```

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

The following code displays the named options:

C#

```
public class TestNOModel : PageModel
{
    private readonly TopItemSettings _monthTopItem;
    private readonly TopItemSettings _yearTopItem;

    public TestNOModel(IOptionsSnapshot<TopItemSettings>
namedOptionsAccessor)
    {
        _monthTopItem = namedOptionsAccessor.Get(TopItemSettings.Month);
        _yearTopItem = namedOptionsAccessor.Get(TopItemSettings.Year);
    }

    public ContentResult OnGet()
    {
        return Content($"Month:Name {_monthTopItem.Name} \n" +
                       $"Month:Model {_monthTopItem.Model} \n\n" +
                       $"Year:Name {_yearTopItem.Name} \n" +
                       $"Year:Model {_yearTopItem.Model} \n"    );
    }
}
```

All options are named instances. IConfigureOptions<TOptions> instances are treated as targeting the `Options.DefaultName` instance, which is `string.Empty`. IConfigureNamedOptions<TOptions> also implements IConfigureOptions<TOptions>. The default implementation of the IOptionsFactory<TOptions> has logic to use each appropriately. The `null` named option is used to target all of the named instances instead of a specific named instance. ConfigureAll and PostConfigureAll use this convention.

OptionsBuilder API

OptionsBuilder<TOptions> is used to configure `TOptions` instances. `OptionsBuilder` streamlines creating named options as it's only a single parameter to the initial `AddOptions<TOptions>(string optionsName)` call instead of appearing in all of the subsequent calls. Options validation and the `ConfigureOptions` overloads that accept service dependencies are only available via `OptionsBuilder`.

`OptionsBuilder` is used in the Options validation section.

See Use AddOptions to configure custom repository for information adding a custom repository.

Use DI services to configure options

Services can be accessed from dependency injection while configuring options in two ways:

- Pass a configuration delegate to Configure on OptionsBuilder<TOptions>. `OptionsBuilder<TOptions>` provides overloads of Configure that allow use of up to five services to configure options:

  C#

  ```
  builder.Services.AddOptions<MyOptions>("optionalName")
      .Configure<Service1, Service2, Service3, Service4, Service5>(
          (o, s, s2, s3, s4, s5) =>
              o.Property = DoSomethingWith(s, s2, s3, s4, s5));
  ```

- Create a type that implements IConfigureOptions<TOptions> or IConfigureNamedOptions<TOptions> and register the type as a service.

We recommend passing a configuration delegate to Configure, since creating a service is more complex. Creating a type is equivalent to what the framework does when calling Configure. Calling Configure registers a transient generic IConfigureNamedOptions<TOptions>, which has a constructor that accepts the generic service types specified.

Options validation

Options validation enables option values to be validated.

Consider the following `appsettings.json` file:

JSON

```
{
  "MyConfig": {
    "Key1": "My Key One",
    "Key2": 10,
    "Key3": 32
  }
}
```

The following class is used to bind to the `"MyConfig"` configuration section and applies a couple of `DataAnnotations` rules:

C#

```
public class MyConfigOptions
{
    public const string MyConfig = "MyConfig";

    [RegularExpression(@"^[a-zA-Z''-'\s]{1,40}$")]
    public string Key1 { get; set; }
    [Range(0, 1000,
        ErrorMessage = "Value for {0} must be between {1} and {2}.")]
    public int Key2 { get; set; }
    public int Key3 { get; set; }
}
```

The following code:

- Calls AddOptions to get an OptionsBuilder<TOptions> that binds to the `MyConfigOptions` class.
- Calls ValidateDataAnnotations to enable validation using `DataAnnotations`.

C#

```
using OptionsValidationSample.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddOptions<MyConfigOptions>()

.Bind(builder.Configuration.GetSection(MyConfigOptions.MyConfig))
            .ValidateDataAnnotations();

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

The `ValidateDataAnnotations` extension method is defined in the Microsoft.Extensions.Options.DataAnnotations NuGet package. For web apps that use the `Microsoft.NET.Sdk.Web` SDK, this package is referenced implicitly from the shared framework.

The following code displays the configuration values or the validation errors:

C#

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly IOptions<MyConfigOptions> _config;

    public HomeController(IOptions<MyConfigOptions> config,
                          ILogger<HomeController> logger)
    {
        _config = config;
        _logger = logger;

        try
        {
            var configValue = _config.Value;

        }
        catch (OptionsValidationException ex)
        {
            foreach (var failure in ex.Failures)
            {
                _logger.LogError(failure);
            }
        }
    }

    public ContentResult Index()
    {
        string msg;
        try
        {
            msg = $"Key1: {_config.Value.Key1} \n" +
                  $"Key2: {_config.Value.Key2} \n" +
                  $"Key3: {_config.Value.Key3}";
        }
        catch (OptionsValidationException optValEx)
        {
            return Content(optValEx.Message);
        }
        return Content(msg);
    }
```

The following code applies a more complex validation rule using a delegate:

C#

```
using OptionsValidationSample.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddOptions<MyConfigOptions>()
```

```
.Bind(builder.Configuration.GetSection(MyConfigOptions.MyConfig))
            .ValidateDataAnnotations()
        .Validate(config =>
        {
            if (config.Key2 != 0)
            {
                return config.Key3 > config.Key2;
            }

            return true;
        }, "Key3 must be > than Key2.");   // Failure message.

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

`IValidateOptions<TOptions>` and `IValidatableObject`

The following class implements IValidateOptions<TOptions>:

C#

```
public class MyConfigValidation : IValidateOptions<MyConfigOptions>
{
    public MyConfigOptions _config { get; private set; }

    public  MyConfigValidation(IConfiguration config)
    {
        _config = config.GetSection(MyConfigOptions.MyConfig)
            .Get<MyConfigOptions>();
    }

    public ValidateOptionsResult Validate(string name, MyConfigOptions
options)
    {
        string? vor = null;
        var rx = new Regex(@"^[a-zA-Z''-'\s]{1,40}$");
        var match = rx.Match(options.Key1!);

        if (string.IsNullOrEmpty(match.Value))
        {
            vor = $"{options.Key1} doesn't match RegEx \n";
        }

        if ( options.Key2 < 0 || options.Key2 > 1000)
        {
            vor = $"{options.Key2} doesn't match Range 0 - 1000 \n";
        }

        if (_config.Key2 != default)
        {
            if(_config.Key3 <= _config.Key2)
            {
```

```
                vor +=  "Key3 must be > than Key2.";
            }
        }

        if (vor != null)
        {
            return ValidateOptionsResult.Fail(vor);
        }

        return ValidateOptionsResult.Success;
    }
}
```

`IValidateOptions` enables moving the validation code out of `Program.cs` and into a class.

Using the preceding code, validation is enabled in `Program.cs` with the following code:

C#

```
using Microsoft.Extensions.Options;
using OptionsValidationSample.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.Configure<MyConfigOptions>
(builder.Configuration.GetSection(
                                        MyConfigOptions.MyConfig));

builder.Services.AddSingleton<IValidateOptions
                              <MyConfigOptions>, MyConfigValidation>();

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

Options validation also supports IValidatableObject. To perform class-level validation of a class within the class itself:

- Implement the `IValidatableObject` interface and its Validate method within the class.
- Call ValidateDataAnnotations in `Program.cs`.

`ValidateOnStart`

Options validation runs the first time a `TOption` instance is created. That means, for instance, when first access to IOptionsSnapshot<TOptions>.Value occurs in a request pipeline or when IOptionsMonitor<TOptions>.Get(string) is called on settings present.

After settings are reloaded, validation runs again. The ASP.NET Core runtime uses OptionsCache<TOptions> to cache the options instance once it is created.

To run options validation eagerly, when the app starts, call ValidateOnStart<TOptions>(OptionsBuilder<TOptions>)in `Program.cs`:

C#

```
builder.Services.AddOptions<MyConfigOptions>()
    .Bind(builder.Configuration.GetSection(MyConfigOptions.MyConfig))
    .ValidateDataAnnotations()
    .ValidateOnStart();
```

Options post-configuration

Set post-configuration with IPostConfigureOptions<TOptions>. Post-configuration runs after all IConfigureOptions<TOptions> configuration occurs:

C#

```
using OptionsValidationSample.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddOptions<MyConfigOptions>()

.Bind(builder.Configuration.GetSection(MyConfigOptions.MyConfig));

builder.Services.PostConfigure<MyConfigOptions>(myOptions =>
{
    myOptions.Key1 = "post_configured_key1_value";
});
```

PostConfigure is available to post-configure named options:

C#

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

builder.Services.AddRazorPages();

builder.Services.Configure<TopItemSettings>(TopItemSettings.Month,
    builder.Configuration.GetSection("TopItem:Month"));
builder.Services.Configure<TopItemSettings>(TopItemSettings.Year,
    builder.Configuration.GetSection("TopItem:Year"));
```

```
builder.Services.PostConfigure<TopItemSettings>("Month", myOptions =>
{
    myOptions.Name = "post_configured_name_value";
    myOptions.Model = "post_configured_model_value";
});

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

Use PostConfigureAll to post-configure all configuration instances:

C#

```
using OptionsValidationSample.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddOptions<MyConfigOptions>()

.Bind(builder.Configuration.GetSection(MyConfigOptions.MyConfig));

builder.Services.PostConfigureAll<MyConfigOptions>(myOptions =>
{
    myOptions.Key1 = "post_configured_key1_value";
});
```

Access options in `Program.cs`

To access IOptions<TOptions> or IOptionsMonitor<TOptions> in `Program.cs`, call GetRequiredService on WebApplication.Services:

C#

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

var option1 = app.Services.GetRequiredService<IOptionsMonitor<MyOptions>>()
    .CurrentValue.Option1;
```

Additional resources

- View or download sample code (how to download)

Use multiple environments in ASP.NET Core

Article • 09/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Rick Anderson and Kirk Larkin

ASP.NET Core configures app behavior based on the runtime environment using an environment variable.

For Blazor environments guidance, which adds to or supersedes the guidance in this article, see ASP.NET Core Blazor environments.

Environments

To determine the runtime environment, ASP.NET Core reads from the following environment variables:

1. DOTNET_ENVIRONMENT
2. `ASPNETCORE_ENVIRONMENT` when the WebApplication.CreateBuilder method is called. The default ASP.NET Core web app templates call `WebApplication.CreateBuilder`. The `DOTNET_ENVIRONMENT` value overrides `ASPNETCORE_ENVIRONMENT` when `WebApplicationBuilder` is used. For other hosts, such as `ConfigureWebHostDefaults` and `WebHost.CreateDefaultBuilder`, `ASPNETCORE_ENVIRONMENT` has higher precedence.

`IHostEnvironment.EnvironmentName` can be set to any value, but the following values are provided by the framework:

- Development: The launchSettings.json file sets `ASPNETCORE_ENVIRONMENT` to `Development` on the local machine.
- Staging

- Production: The default if `DOTNET_ENVIRONMENT` and `ASPNETCORE_ENVIRONMENT` have not been set.

The following code:

- Is similar to the code generated by the ASP.NET Core templates.
- Enables the Developer Exception Page when `ASPNETCORE_ENVIRONMENT` is set to `Development`. This is done automatically by the WebApplication.CreateBuilder method.
- Calls UseExceptionHandler when the value of `ASPNETCORE_ENVIRONMENT` is anything other than `Development`.
- Provides an IWebHostEnvironment instance in the Environment property of `WebApplication`.

C#

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

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

var app = builder.Build();

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

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

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The Environment Tag Helper uses the value of IHostEnvironment.EnvironmentName to include or exclude markup in the element:

CSHTML

```
<environment include="Development">
    <div>Environment is Development</div>
</environment>
<environment exclude="Development">
    <div>Environment is NOT Development</div>
</environment>
<environment include="Staging,Development,Staging_2">
    <div>Environment is: Staging, Development or Staging_2</div>
</environment>
```

The About page from the sample code includes the preceding markup and displays the value of `IWebHostEnvironment.EnvironmentName`.

On Windows and macOS, environment variables and values aren't case-sensitive. Linux environment variables and values are case-sensitive by default.

Create EnvironmentsSample

The sample code used in this article is based on a Razor Pages project named *EnvironmentsSample*.

The following .NET CLI commands create and run a web app named *EnvironmentsSample*:

Bash

```
dotnet new webapp -o EnvironmentsSample
cd EnvironmentsSample
dotnet run --verbosity normal
```

When the app runs, it displays output similar to the following:

Bash

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7152
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5105
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Path\To\EnvironmentsSample
```

Set environment on the command line

Use the `--environment` flag to set the environment. For example:

.NET CLI

```
dotnet run --environment Production
```

The preceding command sets the environment to `Production` and displays output similar to the following in the command window:

Bash

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7262
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5005
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Path\To\EnvironmentsSample
```

Development and launchSettings.json

The development environment can enable features that shouldn't be exposed in production. For example, the ASP.NET Core project templates enable the Developer Exception Page in the development environment. Because of the performance cost, scope validation and dependency validation only happens in development.

The environment for local machine development can be set in the *Properties\launchSettings.json* file of the project. Environment values set in `launchSettings.json` override values set in the system environment.

The `launchSettings.json` file:

- Is only used on the local development machine.
- Is not deployed.
- Contains profile settings.

The following JSON shows the `launchSettings.json` file for an ASP.NET Core web project named *EnvironmentsSample* created with Visual Studio or `dotnet new`:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:59481",
      "sslPort": 44308
    }
  },
  "profiles": {
    "EnvironmentsSample": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7152;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

The preceding JSON contains two profiles:

- `EnvironmentsSample`: The profile name is the project name. As the first profile listed, this profile is used by default. The `"commandName"` key has the value `"Project"`, therefore, the Kestrel web server is launched.
- `IIS Express`: The `"commandName"` key has the value `"IISExpress"`, therefore, IISExpress is the web server.

You can set the launch profile to the project or any other profile included in `launchSettings.json`. For example, in the image below, selecting the project name launches the Kestrel web server.