

The value of `commandName` can specify the web server to launch. `commandName` can be any one of the following:

- `IISExpress` : Launches IIS Express.
- `IIS` : No web server launched. IIS is expected to be available.
- `Project` : Launches Kestrel.

The Visual Studio 2022 project properties **Debug / General** tab provides an **Open debug launch profiles UI** link. This link opens a **Launch Profiles** dialog that lets you edit the environment variable settings in the `launchSettings.json` file. You can also open the **Launch Profiles** dialog from the **Debug** menu by selecting **<project name> Debug Properties**. Changes made to project profiles may not take effect until the web server is restarted. Kestrel must be restarted before it can detect changes made to its environment.

**Launch Profiles**

- EnvironmentsSample
- EnvironmentsSample-Staging
- EnvironmentsSample-Production
- IIS Express

**Command line arguments**  
Command line arguments to pass to the executable.

**Working directory**  
Path to the working directory where the process will be started.

**Environment variables**  
The environment variables to set prior to running the process. Variables are separated by a comma (,) and variables names and values are separated with an equal sign (=). Example: var1=value1,var2=value2,var3=value3. Commas and equal signs appearing within a variable can be escaped using a forward slash (/).

**Enable Hot Reload**  
☒ Apply code changes to the running application.

**Enable native code debugging**

The following `launchSettings.json` file contains multiple profiles:

## 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"
      }
    },
    "EnvironmentsSample-Staging": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7152;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging",
        "ASPNETCORE_DETAILEDERRORS": "1",

```

```

        "ASPNETCORE_SHUTDOWNTIMEOUTSECONDS": "3"
    },
    },
    "EnvironmentsSample-Production": {
        "commandName": "Project",
        "dotnetRunMessages": true,
        "launchBrowser": true,
        "applicationUrl": "https://localhost:7152;http://localhost:5105",
        "environmentVariables": {
            "ASPNETCORE_ENVIRONMENT": "Production"
        }
    },
    "IIS Express": {
        "commandName": "IISExpress",
        "launchBrowser": true,
        "environmentVariables": {
            "ASPNETCORE_ENVIRONMENT": "Development"
        }
    }
}
}
}

```

Profiles can be selected:

- From the Visual Studio UI.
- Using the `dotnet run` CLI command with the `--launch-profile` option set to the profile's name. *This approach only supports Kestrel profiles.*

.NET CLI

```
dotnet run --launch-profile "EnvironmentsSample"
```

### ⚠ Warning

`launchSettings.json` shouldn't store secrets. The [Secret Manager tool](#) can be used to store secrets for local development.

When using [Visual Studio Code](#), environment variables can be set in the `.vscode/launch.json` file. The following example sets several [environment variables for Host configuration values](#):

JSON

```

{
    "version": "0.2.0",
    "configurations": [

```

```
{
  "name": ".NET Core Launch (web)",
  "type": "coreclr",
  // Configuration omitted for brevity.
  "env": {
    "ASPNETCORE_ENVIRONMENT": "Development",
    "ASPNETCORE_URLS": "https://localhost:5001",
    "ASPNETCORE_DETAILEDERRORS": "1",
    "ASPNETCORE_SHUTDOWNTIMEOUTSECONDS": "3"
  },
  // Configuration omitted for brevity.
}
```

The `.vscode/launch.json` file is used only by Visual Studio Code.

## Production

The production environment should be configured to maximize security, [performance](#), and application robustness. Some common settings that differ from development include:

- [Caching](#).
- Client-side resources are bundled, minified, and potentially served from a CDN.
- Diagnostic error pages disabled.
- Friendly error pages enabled.
- Production [logging](#) and monitoring enabled. For example, using [Application Insights](#).

## Set the environment by setting an environment variable

It's often useful to set a specific environment for testing with an environment variable or platform setting. If the environment isn't set, it defaults to `Production`, which disables most debugging features. The method for setting the environment depends on the operating system.

When the host is built, the last environment setting read by the app determines the app's environment. The app's environment can't be changed while the app is running.

The [About page](#) [↗](#) from the [sample code](#) [↗](#) displays the value of

```
IWebHostEnvironment.EnvironmentName.
```

## Azure App Service

**Production** is the default value if `DOTNET_ENVIRONMENT` and `ASPNETCORE_ENVIRONMENT` have not been set. Apps deployed to Azure are **Production** by default.

To set the environment in an [Azure App Service](#) app by using the portal:

1. Select the app from the **App Services** page.
2. In the **Settings** group, select **Environment variables**.
3. In the **App settings** tab, select **+ Add**.
4. In the **Add/Edit application setting** window, provide `ASPNETCORE_ENVIRONMENT` for the **Name**. For **Value**, provide the environment (for example, `Staging`).
5. Select the **Deployment slot setting** checkbox if you wish the environment setting to remain with the current slot when deployment slots are swapped. For more information, see [Set up staging environments in Azure App Service](#) in the Azure documentation.
6. Select **OK** to close the **Add/Edit application setting** dialog.
7. Select **Save** at the top of the **Configuration** page.

Azure App Service automatically restarts the app after an app setting is added, changed, or deleted in the Azure portal.

## Windows - Set environment variable for a process

Environment values in `launchSettings.json` override values set in the system environment.

To set the `ASPNETCORE_ENVIRONMENT` for the current session when the app is started using [dotnet run](#), use the following commands at a command prompt or in PowerShell:

Console

```
set ASPNETCORE_ENVIRONMENT=Staging
dotnet run --no-launch-profile
```

PowerShell

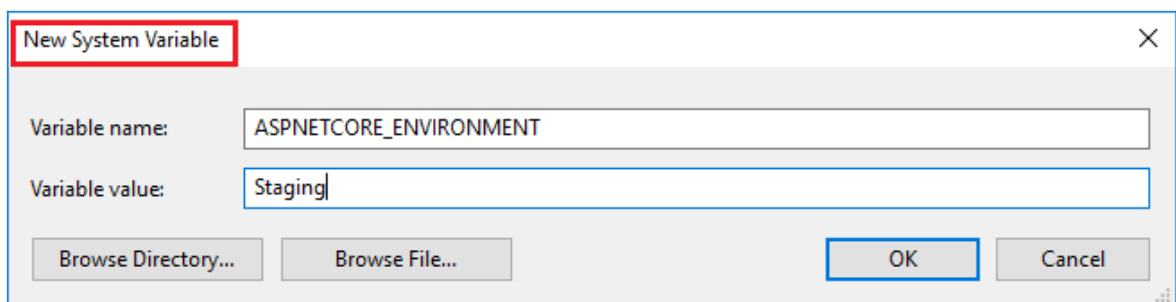
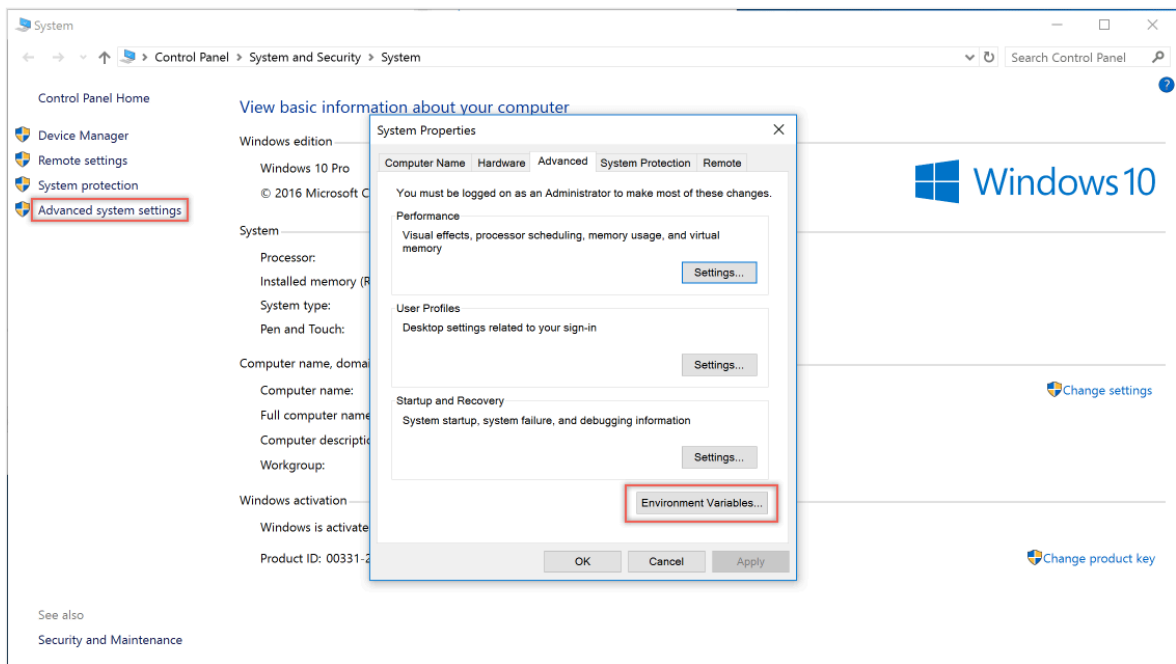
```
$Env:ASPNETCORE_ENVIRONMENT = "Staging"
dotnet run --no-launch-profile
```

## Windows - Set environment variable globally

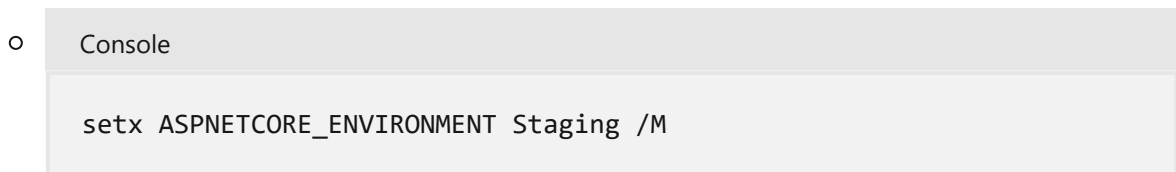
The preceding commands set `ASPNETCORE_ENVIRONMENT` only for processes launched from that command window.

To set the value globally in Windows, use either of the following approaches:

- Open the **Control Panel > System > Advanced system settings** and add or edit the `ASPNETCORE_ENVIRONMENT` value:



- Open an administrative command prompt and use the `setx` command or open an administrative PowerShell command prompt and use `[Environment]::SetEnvironmentVariable:`



The `/M` switch sets the environment variable at the system level. If the `/M` switch isn't used, the environment variable is set for the user account.



The `Machine` option sets the environment variable at the system level. If the option value is changed to `User`, the environment variable is set for the user account.

When the `ASPNETCORE_ENVIRONMENT` environment variable is set globally, it takes effect for `dotnet run` in any command window opened after the value is set. Environment values in `launchSettings.json` override values set in the system environment.

## Windows - Use web.config

To set the `ASPNETCORE_ENVIRONMENT` environment variable with `web.config`, see the *Set environment variables* section of [web.config file](#).

## Windows - IIS deployments

Include the `<EnvironmentName>` property in the [publish profile \(.pubxml\)](#) or project file. This approach sets the environment in `web.config` when the project is published:

XML

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

To set the `ASPNETCORE_ENVIRONMENT` environment variable for an app running in an isolated Application Pool (supported on IIS 10.0 or later), see the *AppCmd.exe command* section of [Environment Variables <environmentVariables>](#). When the `ASPNETCORE_ENVIRONMENT` environment variable is set for an app pool, its value overrides a setting at the system level.

When hosting an app in IIS and adding or changing the `ASPNETCORE_ENVIRONMENT` environment variable, use one of the following approaches to have the new value picked up by apps:

- Execute `net stop was /y` followed by `net start w3svc` from a command prompt.
- Restart the server.

## macOS

Setting the current environment for macOS can be performed in-line when running the app:

---

Bash

```
ASPNETCORE_ENVIRONMENT=Staging dotnet run
```

Alternatively, set the environment with `export` prior to running the app:

Bash

```
export ASPNETCORE_ENVIRONMENT=Staging
```

Machine-level environment variables are set in the `.bashrc` or `.bash_profile` file. Edit the file using any text editor. Add the following statement:

Bash

```
export ASPNETCORE_ENVIRONMENT=Staging
```

## Linux

For Linux distributions, use the `export` command at a command prompt for session-based variable settings and the `bash_profile` file for machine-level environment settings.

## Set the environment in code

To set the environment in code, use `WebApplicationOptions.EnvironmentName` when creating `WebApplicationBuilder`, as shown in the following example:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    EnvironmentName = Environments.Staging
});

// 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();
```

## Configuration by environment

To load configuration by environment, see [Configuration in ASP.NET Core](#).

## Configure services and middleware by environment

Use [WebApplicationBuilder.Environment](#) or [WebApplication.Environment](#) to conditionally add services or middleware depending on the current environment. The project template includes an example of code that adds middleware only when the current environment isn't Development:

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 highlighted code checks the current environment while building the request pipeline. To check the current environment while configuring services, use `builder.Environment` instead of `app.Environment`.

## Additional resources

- [View or download sample code](#) <sup>↗</sup> (how to download)
- [App startup in ASP.NET Core](#)
- [Configuration in ASP.NET Core](#)
- [ASP.NET Core Blazor environments](#)

# Logging in .NET Core and 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 [Kirk Larkin](#), [Juergen Gutsch](#), and [Rick Anderson](#)

This article describes logging in .NET as it applies to ASP.NET Core apps. For detailed information on logging in .NET, see [Logging in .NET](#).

For Blazor logging guidance, which adds to or supersedes the guidance in this node, see [ASP.NET Core Blazor logging](#).

## Logging providers

Logging providers store logs, except for the `Console` provider which displays logs. For example, the Azure Application Insights provider stores logs in [Azure Application Insights](#). Multiple providers can be enabled.

The default ASP.NET Core web app templates call [WebApplication.CreateBuilder](#), which adds the following logging providers:

- [Console](#)
- [Debug](#)
- [EventSource](#)
- [EventLog](#): Windows only

C#

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

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
```

```

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

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

    app.UseRouting();

    app.UseAuthorization();

    app.MapRazorPages();

    app.Run();

```

The preceding code shows the `Program.cs` file created with the ASP.NET Core web app templates. The next several sections provide samples based on the ASP.NET Core web app templates.

The following code overrides the default set of logging providers added by `WebApplication.CreateBuilder`:

```

C#

var builder = WebApplication.CreateBuilder(args);
builder.Logging.ClearProviders();
builder.Logging.AddConsole();

builder.Services.AddRazorPages();

var app = builder.Build();

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

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

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();

```

Alternatively, the preceding logging code can be written as follows:

C#

```
var builder = WebApplication.CreateBuilder();
builder.Host.ConfigureLogging(logging =>
{
    logging.ClearProviders();
    logging.AddConsole();
});
```

For additional providers, see:

- [Built-in logging providers](#)
- [Third-party logging providers](#).

## Create logs

To create logs, use an `ILogger<TCategoryName>` object from [dependency injection \(DI\)](#).

The following example:

- Creates a logger, `ILogger<AboutModel>`, which uses a log *category* of the fully qualified name of the type `AboutModel`. The log category is a string that is associated with each log.
- Calls [LogInformation](#) to log at the [Information](#) level. The Log *level* indicates the severity of the logged event.

C#

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

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

    public void OnGet()
    {
        _logger.LogInformation("About page visited at {DT}",
            DateTime.UtcNow.ToLongTimeString());
    }
}
```

[Levels](#) and [categories](#) are explained in more detail later in this document.

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

# Configure logging

Logging configuration is commonly provided by the `Logging` section of `appsettings.{ENVIRONMENT}.json` files, where the `{ENVIRONMENT}` placeholder is the [environment](#). The following `appsettings.Development.json` file is generated by the ASP.NET Core web app templates:

JSON

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

In the preceding JSON:

- The `"Default"` and `"Microsoft.AspNetCore"` categories are specified.
- The `"Microsoft.AspNetCore"` category applies to all categories that start with `"Microsoft.AspNetCore"`. For example, this setting applies to the `"Microsoft.AspNetCore.Routing.EndpointMiddleware"` category.
- The `"Microsoft.AspNetCore"` category logs at log level `Warning` and higher.
- A specific log provider is not specified, so `LogLevel` applies to all the enabled logging providers except for the [Windows EventLog](#).

The `Logging` property can have `LogLevel` and log provider properties. The `LogLevel` specifies the minimum [level](#) to log for selected categories. In the preceding JSON, `Information` and `Warning` log levels are specified. `LogLevel` indicates the severity of the log and ranges from 0 to 6:

`Trace` = 0, `Debug` = 1, `Information` = 2, `Warning` = 3, `Error` = 4, `Critical` = 5, and `None` = 6.

When a `LogLevel` is specified, logging is enabled for messages at the specified level and higher. In the preceding JSON, the `Default` category is logged for `Information` and higher. For example, `Information`, `Warning`, `Error`, and `Critical` messages are logged. If no `LogLevel` is specified, logging defaults to the `Information` level. For more information, see [Log levels](#).

A provider property can specify a `LogLevel` property. `LogLevel` under a provider specifies levels to log for that provider, and overrides the non-provider log settings. Consider the following `appsettings.json` file:

JSON

```
{
  "Logging": {
    "LogLevel": { // All providers, LogLevel applies to all the enabled
providers.
      "Default": "Error", // Default logging, Error and higher.
      "Microsoft": "Warning" // All Microsoft* categories, Warning and
higher.
    },
    "Debug": { // Debug provider.
      "LogLevel": {
        "Default": "Information", // Overrides preceding LogLevel:Default
setting.
        "Microsoft.Hosting": "Trace" // Debug:Microsoft.Hosting category.
      }
    },
    "EventSource": { // EventSource provider
      "LogLevel": {
        "Default": "Warning" // All categories of EventSource provider.
      }
    }
  }
}
```

Settings in `Logging.{PROVIDER NAME}.LogLevel` override settings in `Logging.LogLevel`, where the `{PROVIDER NAME}` placeholder is the provider name. In the preceding JSON, the `Debug` provider's default log level is set to `Information`:

`Logging:Debug:LogLevel:Default:Information`

The preceding setting specifies the `Information` log level for every `Logging:Debug:` category except `Microsoft.Hosting`. When a specific category is listed, the specific category overrides the default category. In the preceding JSON, the `Logging:Debug:LogLevel` categories `"Microsoft.Hosting"` and `"Default"` override the settings in `Logging:LogLevel`.

The minimum log level can be specified for any of:

- Specific providers: For example, `Logging:EventSource:LogLevel:Default:Information`
- Specific categories: For example, `Logging:LogLevel:Microsoft:Warning`
- All providers and all categories: `Logging:LogLevel:Default:Warning`

Any logs below the minimum level are *not*:

- Passed to the provider.
- Logged or displayed.

To suppress all logs, specify `LogLevel.None`. `LogLevel.None` has a value of 6, which is higher than `LogLevel.Critical` (5).

If a provider supports [log scopes](#), `IncludeScopes` indicates whether they're enabled. For more information, see [log scopes](#).

The following `appsettings.json` file contains all the providers enabled by default:

JSON

```
{
  "Logging": {
    "LogLevel": { // No provider, LogLevel applies to all the enabled
providers.
      "Default": "Error",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Warning"
    },
    "Debug": { // Debug provider.
      "LogLevel": {
        "Default": "Information" // Overrides preceding LogLevel:Default
setting.
      }
    },
  },
  "Console": {
    "IncludeScopes": true,
    "LogLevel": {
      "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
      "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
      "Microsoft.AspNetCore.Mvc.Razor": "Error",
      "Default": "Information"
    }
  },
  "EventSource": {
    "LogLevel": {
      "Microsoft": "Information"
    }
  },
  "EventLog": {
    "LogLevel": {
      "Microsoft": "Information"
    }
  },
  "AzureAppServicesFile": {
    "IncludeScopes": true,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```



```

    }
  },
  "AzureAppServicesBlob": {
    "IncludeScopes": true,
    "LogLevel": {
      "Microsoft": "Information"
    }
  },
  "ApplicationInsights": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
}
}

```

In the preceding sample:

- The categories and levels aren't suggested values. The sample is provided to show all of the default providers.
- Settings in `Logging.{PROVIDER NAME}.LogLevel` override settings in `Logging.LogLevel`, where the `{PROVIDER NAME}` placeholder is the provider name. For example, the level in `Debug.LogLevel.Default` overrides the level in `LogLevel.Default`.
- Each default provider *alias* is used. Each provider defines an *alias* that can be used in configuration in place of the fully qualified type name. The built-in providers aliases are:
  - `Console`
  - `Debug`
  - `EventSource`
  - `EventLog`
  - `AzureAppServicesFile`
  - `AzureAppServicesBlob`
  - `ApplicationInsights`

## Log in Program.cs

The following example calls `Builder.WebApplication.Logger` in `Program.cs` and logs informational messages:

C#

```

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

```

```
app.Logger.LogInformation("Adding Routes");
app.MapGet("/", () => "Hello World!");
app.Logger.LogInformation("Starting the app");
app.Run();
```

The following example calls `AddConsole` in `Program.cs` and logs the `/Test` endpoint:

C#

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

builder.Logging.AddConsole();

var app = builder.Build();

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

app.MapGet("/Test", async (ILogger<Program> logger, HttpResponse response)
=>
{
    logger.LogInformation("Testing logging in Program.cs");
    await response.WriteAsync("Testing");
});

app.Run();
```

The following example calls `AddSimpleConsole` in `Program.cs`, disables color output, and logs the `/Test` endpoint:

C#

```
using Microsoft.Extensions.Logging.Console;

var builder = WebApplication.CreateBuilder(args);

builder.Logging.AddSimpleConsole(i => i.ColorBehavior =
LoggerColorBehavior.Disabled);

var app = builder.Build();

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

app.MapGet("/Test", async (ILogger<Program> logger, HttpResponse response)
=>
{
    logger.LogInformation("Testing logging in Program.cs");
    await response.WriteAsync("Testing");
});

app.Run();
```

# Set log level by command line, environment variables, and other configuration

Log level can be set by any of the [configuration providers](#).

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 key `Logging:LogLevel:Microsoft` to a value of `Information` on Windows.
- Test the settings when using an app created with the ASP.NET Core web application templates. The `dotnet run` command must be run in the project directory after using `set`.

.NET CLI

```
set Logging__LogLevel__Microsoft=Information
dotnet run
```

The preceding environment setting:

- Is only set in processes launched from the command window they were set in.
- Isn't read by browsers launched with Visual Studio.

The following `setx` command also sets the environment key and value on Windows. Unlike `set`, `setx` settings are persisted. The `/M` switch sets the variable in the system environment. If `/M` isn't used, a user environment variable is set.

Console

```
setx Logging__LogLevel__Microsoft Information /M
```

Consider the following `appsettings.json` file:

JSON

```
"Logging": {  
  "Console": {  
    "LogLevel": {  
      "Microsoft.Hosting.Lifetime": "Trace"  
    }  
  }  
}
```

The following command sets the preceding configuration in the environment:

Console

```
setx Logging__Console__LogLevel__Microsoft.Hosting.Lifetime Trace /M
```

### ⓘ Note

When configuring environment variables with names that contain `.` (periods) in macOS and Linux, consider the "Exporting a variable with a dot (.) in it" question on [Stack Exchange](#) and its corresponding [accepted answer](#).

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](#).

For more information on setting ASP.NET Core configuration values using environment variables, see [environment variables](#). For information on using other configuration sources, including the command line, Azure Key Vault, Azure App Configuration, other file formats, and more, see [Configuration in ASP.NET Core](#).

## How filtering rules are applied

When an `ILogger<TCategoryName>` object is created, the `ILoggerFactory` object selects a single rule per provider to apply to that logger. All messages written by an `ILogger` instance are filtered based on the selected rules. The most specific rule for each provider and category pair is selected from the available rules.

The following algorithm is used for each provider when an `ILogger` is created for a given category:

- Select all rules that match the provider or its alias. If no match is found, select all rules with an empty provider.
- From the result of the preceding step, select rules with longest matching category prefix. If no match is found, select all rules that don't specify a category.
- If multiple rules are selected, take the **last** one.
- If no rules are selected, use `MinimumLevel`.

## Logging output from dotnet run and Visual Studio

Logs created with the [default logging providers](#) are displayed:

- In Visual Studio
  - In the Debug output window when debugging.
  - In the ASP.NET Core Web Server window.
- In the console window when the app is run with `dotnet run`.

Logs that begin with "Microsoft" categories are from .NET. .NET and application code use the same [logging API and providers](#).

## Log category

When an `ILogger` object is created, a *category* is specified. That category is included with each log message created by that instance of `ILogger`. The category string is arbitrary, but the convention is to use the fully qualified class name. For example, in a controller the name might be `"TodoApi.Controllers.TODOController"`. The ASP.NET Core web apps use `ILogger<T>` to automatically get an `ILogger` instance that uses the fully qualified type name of `T` as the category:

C#

```
public class PrivacyModel : PageModel
{
    private readonly ILogger<PrivacyModel> _logger;

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

```

public void OnGet()
{
    _logger.LogInformation("GET Pages.PrivacyModel called.");
}
}

```

If further categorization is desired, the convention is to use a hierarchical name by appending a subcategory to the fully qualified class name, and explicitly specify the category using `ILoggerFactory.CreateLogger`:

C#

```

public class ContactModel : PageModel
{
    private readonly ILogger _logger;

    public ContactModel(ILoggerFactory logger)
    {
        _logger =
logger.CreateLogger("TodoApi.Pages.ContactModel.MyCategory");
    }

    public void OnGet()
    {
        _logger.LogInformation("GET Pages.ContactModel called.");
    }
}

```

Calling `CreateLogger` with a fixed name can be useful when used in multiple methods so the events can be organized by category.

`ILogger<T>` is equivalent to calling `CreateLogger` with the fully qualified type name of `T`.

## Log level

The following table lists the `LogLevel` values, the convenience `Log{LogLevel}` extension method, and the suggested usage:

[Expand table](#)

LogLevel	Value	Method	Description
Trace	0	<code>LogTrace</code>	Contain the most detailed messages. These messages may contain sensitive app data. These messages are disabled by default and should <i>not</i> be enabled in production.

LogLevel	Value	Method	Description
Debug	1	<a href="#">LogDebug</a>	For debugging and development. Use with caution in production due to the high volume.
Information	2	<a href="#">LogInformation</a>	Tracks the general flow of the app. May have long-term value.
Warning	3	<a href="#">LogWarning</a>	For abnormal or unexpected events. Typically includes errors or conditions that don't cause the app to fail.
Error	4	<a href="#">LogError</a>	For errors and exceptions that cannot be handled. These messages indicate a failure in the current operation or request, not an app-wide failure.
Critical	5	<a href="#">LogCritical</a>	For failures that require immediate attention. Examples: data loss scenarios, out of disk space.
None	6		Specifies that a logging category shouldn't write messages.

In the previous table, the `LogLevel` is listed from lowest to highest severity.

The [Log](#) method's first parameter, [LogLevel](#), indicates the severity of the log. Rather than calling `Log(LogLevel, ...)`, most developers call the [Log{LOG LEVEL}](#) extension methods, where the `{LOG LEVEL}` placeholder is the log level. For example, the following two logging calls are functionally equivalent and produce the same log:

C#

```
[HttpGet]
public IActionResult Test1(int id)
{
    var routeInfo = ControllerContext.ToCtxString(id);

    _logger.Log(LogLevel.Information, MyLogEvents.TestItem, routeInfo);
    _logger.LogInformation(MyLogEvents.TestItem, routeInfo);

    return ControllerContext.MyDisplayRouteInfo();
}
```

`MyLogEvents.TestItem` is the event ID. `MyLogEvents` is part of the sample app and is displayed in the [Log event ID](#) section.

[MyDisplayRouteInfo](#) and [ToCtxString](#) are provided by the [Rick.Docs.Samples.RouteInfo](#) NuGet package. The methods display `Controller` and `Razor Page` route information.

The following code creates `Information` and `Warning` logs:

```
C#

[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TODOItems.FindAsync(id);

    if (todoItem == null)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

In the preceding code, the first `Log{LOG LEVEL}` parameter, `MyLogEvents.GetItem`, is the [Log event ID](#). The second parameter is a message template with placeholders for argument values provided by the remaining method parameters. The method parameters are explained in the [message template](#) section later in this document.

Call the appropriate `Log{LOG LEVEL}` method to control how much log output is written to a particular storage medium. For example:

- In production:
  - Logging at the `Trace`, `Debug`, or `Information` levels produces a high-volume of detailed log messages. To control costs and not exceed data storage limits, log `Trace`, `Debug`, or `Information` level messages to a high-volume, low-cost data store. Consider limiting `Trace`, `Debug`, or `Information` to specific categories.
  - Logging at `Warning` through `Critical` levels should produce few log messages.
    - Costs and storage limits usually aren't a concern.
    - Few logs allow more flexibility in data store choices.
- In development:
  - Set to `Warning`.
  - Add `Trace`, `Debug`, or `Information` messages when troubleshooting. To limit output, set `Trace`, `Debug`, or `Information` only for the categories under investigation.

ASP.NET Core writes logs for framework events. For example, consider the log output for:



- A Razor Pages app created with the ASP.NET Core templates.
- Logging set to `Logging:Console:LogLevel:Microsoft:Information`.
- Navigation to the Privacy page:

#### Console

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 GET https://localhost:5001/Privacy
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint '/Privacy'
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[3]
      Route matched with {page = "/Privacy"}. Executing page /Privacy
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[101]
      Executing handler method DefaultRP.Pages.PrivacyModel.OnGet -
ModelState is Valid
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[102]
      Executed handler method OnGet, returned result .
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[103]
      Executing an implicit handler method - ModelState is Valid
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[104]
      Executed an implicit handler method, returned result
Microsoft.AspNetCore.Mvc.RazorPages.PageResult.
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[4]
      Executed page /Privacy in 74.5188ms
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint '/Privacy'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 149.3023ms 200 text/html; charset=utf-8
```

The following JSON sets `Logging:Console:LogLevel:Microsoft:Information`:

#### JSON

```
{
  "Logging": { // Default, all providers.
    "LogLevel": {
      "Microsoft": "Warning"
    },
    "Console": { // Console provider.
      "LogLevel": {
        "Microsoft": "Information"
      }
    }
  }
}
```

# Log event ID

Each log can specify an *event ID*. The sample app uses the `MyLogEvents` class to define event IDs:

C#

```
public class MyLogEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems     = 1001;
    public const int GetItem       = 1002;
    public const int InsertItem    = 1003;
    public const int UpdateItem    = 1004;
    public const int DeleteItem    = 1005;

    public const int TestItem      = 3000;

    public const int GetItemNotFound = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TODOItems.FindAsync(id);

    if (todoItem == null)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

An event ID associates a set of events. For example, all logs related to displaying a list of items on a page might be 1001.

The logging provider may store the event ID in an ID field, in the logging message, or not at all. The Debug provider doesn't show event IDs. The console provider shows event IDs in brackets after the category:

Console

```
info: TodoApi.Controllers.TodoItemsController[1002]  
    Getting item 1  
warn: TodoApi.Controllers.TodoItemsController[4000]  
    Get(1) NOT FOUND
```

Some logging providers store the event ID in a field, which allows for filtering on the ID.

## Log message template

Each log API uses a message template. The message template can contain placeholders for which arguments are provided. Use names for the placeholders, not numbers.

C#

```
[HttpGet("{id}")]  
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)  
{  
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);  
  
    var todoItem = await _context.TodoItems.FindAsync(id);  
  
    if (todoItem == null)  
    {  
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT  
FOUND", id);  
        return NotFound();  
    }  
  
    return ItemToDTO(todoItem);  
}
```

The *order of the parameters*, not their placeholder names, determines which parameters are used to provide placeholder values in log messages. In the following code, the parameter names are out of sequence in the placeholders of the message template:

C#

```
var apples = 1;  
var pears = 2;  
var bananas = 3;  
  
_logger.LogInformation("Parameters: {Pears}, {Bananas}, {Apples}", apples,  
pears, bananas);
```

However, the parameters are assigned to the placeholders in the order: `apples`, `pears`, `bananas`. The log message reflects the *order of the parameters*:

text

Parameters: 1, 2, 3

This approach allows logging providers to implement [semantic or structured logging](#). The arguments themselves are passed to the logging system, not just the formatted message template. This enables logging providers to store the parameter values as fields. For example, consider the following logger method:

C#

```
_logger.LogInformation("Getting item {Id} at {RequestTime}", id,
DateTime.Now);
```

For example, when logging to Azure Table Storage:

- Each Azure Table entity can have `ID` and `RequestTime` properties.
- Tables with properties simplify queries on logged data. For example, a query can find all logs within a particular `RequestTime` range without having to parse the time out of the text message.

## Log exceptions

The logger methods have overloads that take an exception parameter:

C#

```
[HttpGet("{id}")]
public IActionResult TestExp(int id)
{
    var routeInfo = ControllerContext.ToCtxString(id);
    _logger.LogInformation(MyLogEvents.TestItem, routeInfo);

    try
    {
        if (id == 3)
        {
            throw new Exception("Test exception");
        }
    }
    catch (Exception ex)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, ex, "TestExp({Id})",
id);
        return NotFound();
    }
}
```

```
return ControllerContext.MyDisplayRouteInfo();  
}
```

[MyDisplayRouteInfo](#) and [ToCtxString](#) are provided by the [Rick.Docs.Samples.RouteInfo](#) NuGet package. The methods display `Controller` and `Razor Page` route information.

Exception logging is provider-specific.

## Default log level

If the default log level is not set, the default log level value is `Information`.

For example, consider the following web app:

- Created with the ASP.NET web app templates.
- `appsettings.json` and `appsettings.Development.json` deleted or renamed.

With the preceding setup, navigating to the privacy or home page produces many `Trace`, `Debug`, and `Information` messages with `Microsoft` in the category name.

The following code sets the default log level when the default log level is not set in configuration:

C#

```
var builder = WebApplication.CreateBuilder();  
builder.Logging.SetMinimumLevel(LogLevel.Warning);
```

Generally, log levels should be specified in configuration and not code.

## Filter function

A filter function is invoked for all providers and categories that don't have rules assigned to them by configuration or code:

C#

```
var builder = WebApplication.CreateBuilder();  
builder.Logging.AddFilter((provider, category, logLevel) =>  
{  
    if (provider.Contains("ConsoleLoggerProvider")  
        && category.Contains("Controller")  
        && logLevel >= LogLevel.Information)  
    {  
        // Filter logic  
    }  
});
```

```

        return true;
    }
    else if (provider.Contains("ConsoleLoggerProvider")
        && category.Contains("Microsoft")
        && logLevel >= LogLevel.Information)
    {
        return true;
    }
    else
    {
        return false;
    }
});

```

The preceding code displays console logs when the category contains `Controller` or `Microsoft` and the log level is `Information` or higher.

Generally, log levels should be specified in configuration and not code.

## ASP.NET Core categories

The following table contains some categories used by ASP.NET Core.

[Expand table](#)

Category	Notes
<code>Microsoft.AspNetCore</code>	General ASP.NET Core diagnostics.
<code>Microsoft.AspNetCore.DataProtection</code>	Which keys were considered, found, and used.
<code>Microsoft.AspNetCore.HostFiltering</code>	Hosts allowed.
<code>Microsoft.AspNetCore.Hosting</code>	How long HTTP requests took to complete and what time they started. Which hosting startup assemblies were loaded.
<code>Microsoft.AspNetCore.Mvc</code>	MVC and Razor diagnostics. Model binding, filter execution, view compilation, action selection.
<code>Microsoft.AspNetCore.Routing</code>	Route matching information.
<code>Microsoft.AspNetCore.Server</code>	Connection start, stop, and keep alive responses. HTTPS certificate information.
<code>Microsoft.AspNetCore.StaticFiles</code>	Files served.

To view more categories in the console window, set `appsettings.Development.json` to the following:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Trace",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

For a list of Entity Framework categories, see [EF Message categories](#).

## Log scopes

A *scope* can group a set of logical operations. This grouping can be used to attach the same data to each log that's created as part of a set. For example, every log created as part of processing a transaction can include the transaction ID.

A scope:

- Is an [IDisposable](#) type that's returned by the [BeginScope](#) method.
- Lasts until it's disposed.

The following providers support scopes:

- `Console`
- [AzureAppServicesFile](#) and [AzureAppServicesBlob](#)

Use a scope by wrapping logger calls in a `using` block:

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    TodoItem todoItem;
    var transactionId = Guid.NewGuid().ToString();
    using (_logger.BeginScope(new List<KeyValuePair<string, object>>
    {
        new KeyValuePair<string, object>("TransactionId",
transactionId),
    })))
    {
```

```
        _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}",
id);

        todoItem = await _context.TODOItems.FindAsync(id);

        if (todoItem == null)
        {
            _logger.LogWarning(MyLogEvents.GetItemNotFound,
                "Get({Id}) NOT FOUND", id);
            return NotFound();
        }

        return ItemToDTO(todoItem);
    }
}
```

## Built-in logging providers

ASP.NET Core includes the following logging providers as part of the shared framework:

- [Console](#)
- [Debug](#)
- [EventSource](#)
- [EventLog](#)

The following logging providers are shipped by Microsoft, but not as part of the shared framework. They must be installed as additional nuget.

- [AzureAppServicesFile and AzureAppServicesBlob](#)
- [ApplicationInsights](#)

ASP.NET Core doesn't include a logging provider for writing logs to files. To write logs to files from an ASP.NET Core app, consider using a [third-party logging provider](#).

For information on `stdout` and debug logging with the ASP.NET Core Module, see [Troubleshoot ASP.NET Core on Azure App Service and IIS](#) and [ASP.NET Core Module \(ANCM\) for IIS](#).

## Console

The `Console` provider logs output to the console. For more information on viewing `Console` logs in development, see [Logging output from dotnet run and Visual Studio](#).

## Debug



The `Debug` provider writes log output by using the [System.Diagnostics.Debug](#) class. Calls to `System.Diagnostics.Debug.WriteLine` write to the `Debug` provider.

On Linux, the `Debug` provider log location is distribution-dependent and may be one of the following:

- `/var/log/message`
- `/var/log/syslog`

## Event Source

The `EventSource` provider writes to a cross-platform event source with the name `Microsoft-Extensions-Logging`. On Windows, the provider uses [ETW](#).

## dotnet-trace tooling

The [dotnet-trace](#) tool is a cross-platform CLI global tool that enables the collection of .NET Core traces of a running process. The tool collects [Microsoft.Extensions.Logging.EventSource](#) provider data using a [LoggingEventSource](#).

For installation instructions, see [dotnet-trace](#).

Use the `dotnet-trace` tooling to collect a trace from an app:

1. Run the app with the `dotnet run` command.
2. Determine the process identifier (PID) of the .NET Core app:

```
.NET CLI  
  
dotnet-trace ps
```

Find the PID for the process that has the same name as the app's assembly.

3. Execute the `dotnet-trace` command.

General command syntax:

```
.NET CLI  
  
dotnet-trace collect -p {PID}  
  --providers Microsoft-Extensions-Logging:{Keyword}:{Provider Level}  
  :FilterSpecs="  
    {Logger Category 1}:{Category Level 1};  
    {Logger Category 2}:{Category Level 2};
```

```
...
{Logger Category N}:{Category Level N}\"
```

When using a PowerShell command shell, enclose the `--providers` value in single quotes ( ' ):

.NET CLI

```
dotnet-trace collect -p {PID}
  --providers 'Microsoft-Extensions-Logging:{Keyword}:{Provider
Level}
             :FilterSpecs=\"
               {Logger Category 1}:{Category Level 1};
               {Logger Category 2}:{Category Level 2};
               ...
               {Logger Category N}:{Category Level N}\"'
```

On non-Windows platforms, add the `-f speedscope` option to change the format of the output trace file to `speedscope`.

The following table defines the Keyword:

[Expand table](#)

Keyword	Description
1	Log meta events about the <code>LoggingEventSource</code> . Doesn't log events from <code>ILogger</code> .
2	Turns on the <code>Message</code> event when <code>ILogger.Log()</code> is called. Provides information in a programmatic (not formatted) way.
4	Turns on the <code>FormatMessage</code> event when <code>ILogger.Log()</code> is called. Provides the formatted string version of the information.
8	Turns on the <code>MessageJson</code> event when <code>ILogger.Log()</code> is called. Provides a JSON representation of the arguments.

The following table lists the provider levels:

[Expand table](#)

Provider Level	Description
0	<code>LogAlways</code>
1	<code>Critical</code>

Provider Level	Description
2	Error
3	Warning
4	Informational
5	Verbose

The parsing for a category level can be either a string or a number:

[Expand table](#)

Category named value	Numeric value
Trace	0
Debug	1
Information	2
Warning	3
Error	4
Critical	5

The provider level and category level:

- Are in reverse order.
- The string constants aren't all identical.

If no `FilterSpecs` are specified then the `EventSourceLogger` implementation attempts to convert the provider level to a category level and applies it to all categories.

[Expand table](#)

Provider Level	Category Level
Verbose (5)	Debug (1)
Informational (4)	Information (2)
Warning (3)	Warning (3)
Error (2)	Error (4)

Provider Level	Category Level
Critical (1)	Critical (5)

If `FilterSpecs` are provided, any category that is included in the list uses the category level encoded there, all other categories are filtered out.

The following examples assume:

- An app is running and calling `logger.LogDebug("12345")`.
- The process ID (PID) has been set via `set PID=12345`, where `12345` is the actual PID.

Consider the following command:

.NET CLI

```
dotnet-trace collect -p %PID% --providers Microsoft-Extensions-Logging:4:5
```

The preceding command:

- Captures debug messages.
- Doesn't apply a `FilterSpecs`.
- Specifies level 5 which maps category Debug.

Consider the following command:

.NET CLI

```
dotnet-trace collect -p %PID% --providers Microsoft-Extensions-Logging:4:5:\"FilterSpecs=*:5\"
```

The preceding command:

- Doesn't capture debug messages because the category level 5 is `Critical`.
- Provides a `FilterSpecs`.

The following command captures debug messages because category level 1 specifies `Debug`.

.NET CLI

```
dotnet-trace collect -p %PID% --providers Microsoft-Extensions-Logging:4:5:\"FilterSpecs=*:1\"
```

The following command captures debug messages because category specifies `Debug`.

.NET CLI

```
dotnet-trace collect -p %PID% --providers Microsoft-Extensions-Logging:4:5:\"FilterSpecs=*:Debug\"
```

`FilterSpecs` entries for `{Logger Category}` and `{Category Level}` represent additional log filtering conditions. Separate `FilterSpecs` entries with the `;` semicolon character.

Example using a Windows command shell:

.NET CLI

```
dotnet-trace collect -p %PID% --providers Microsoft-Extensions-Logging:4:2:FilterSpecs=\"Microsoft.AspNetCore.Hosting*:4\"
```

The preceding command activates:

- The Event Source logger to produce formatted strings (4) for errors (2).
- `Microsoft.AspNetCore.Hosting` logging at the `Informational` logging level (4).

4. Stop the `dotnet-trace` tooling by pressing the Enter key or `Ctrl` + `C`.

The trace is saved with the name `trace.nettrace` in the folder where the `dotnet-trace` command is executed.

5. Open the trace with [Perfview](#). Open the `trace.nettrace` file and explore the trace events.

If the app doesn't build the host with [WebApplication.CreateBuilder](#), add the Event Source provider to the app's logging configuration.

For more information, see:

- [Trace for performance analysis utility \(dotnet-trace\)](#) (.NET Core documentation)
- [Trace for performance analysis utility \(dotnet-trace\)](#) [↗](#) (dotnet/diagnostics GitHub repository documentation)
- [LoggingEventSource](#)
- [EventLevel](#)
- [Perfview](#): Useful for viewing Event Source traces.

## Perfview

Use the [PerfView utility](#) to collect and view logs. There are other tools for viewing ETW logs, but PerfView provides the best experience for working with the ETW events emitted by ASP.NET Core.

To configure PerfView for collecting events logged by this provider, add the string `*Microsoft-Extensions-Logging` to the **Additional Providers** list. Don't miss the `*` at the start of the string.

## Windows EventLog

The `EventLog` provider sends log output to the Windows Event Log. Unlike the other providers, the `EventLog` provider does *not* inherit the default non-provider settings. If `EventLog` log settings aren't specified, they default to [LogLevel.Warning](#).

To log events lower than [LogLevel.Warning](#), explicitly set the log level. The following example sets the Event Log default log level to [LogLevel.Information](#):

JSON

```
"Logging": {
  "EventLog": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

[AddEventLog](#) overloads can pass in [EventLogSettings](#). If `null` or not specified, the following default settings are used:

- `LogName`: "Application"
- `SourceName`: ".NET Runtime"
- `MachineName`: The local machine name is used.

The following code changes the `SourceName` from the default value of `".NET Runtime"` to `MyLogs`:

C#

```
var builder = WebApplication.CreateBuilder();
builder.Logging.AddEventLog(eventLogSettings =>
{
```

```
eventLogSettings.SourceName = "MyLogs";  
});
```

## Azure App Service

The [Microsoft.Extensions.Logging.AzureAppServices](#) provider package writes logs to text files in an Azure App Service app's file system and to [blob storage](#) in an Azure Storage account.

The provider package isn't included in the shared framework. To use the provider, add the provider package to the project.

To configure provider settings, use [AzureFileLoggerOptions](#) and [AzureBlobLoggerOptions](#), as shown in the following example:

```
C#  
  
using Microsoft.Extensions.Logging.AzureAppServices;  
  
var builder = WebApplication.CreateBuilder();  
builder.Logging.AddAzureWebAppDiagnostics();  
builder.Services.Configure<AzureFileLoggerOptions>(options =>  
{  
    options.FileName = "azure-diagnostics-";  
    options.FileSizeLimit = 50 * 1024;  
    options.RetainedFileCountLimit = 5;  
});  
builder.Services.Configure<AzureBlobLoggerOptions>(options =>  
{  
    options.BlobName = "log.txt";  
});
```

When deployed to Azure App Service, the app uses the settings in the [App Service logs](#) section of the **App Service** page of the Azure portal. When the following settings are updated, the changes take effect immediately without requiring a restart or redeployment of the app.

- **Application Logging (Filesystem)**
- **Application Logging (Blob)**

The default location for log files is in the `D:\home\LogFiles\Application` folder, and the default file name is `diagnostics-yyyymmdd.txt`. The default file size limit is 10 MB, and the default maximum number of files retained is 2. The default blob name is `{app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt`.

This provider only logs when the project runs in the Azure environment.

## Azure log streaming

Azure log streaming supports viewing log activity in real time from:

- The app server
- The web server
- Failed request tracing

To configure Azure log streaming:

- Navigate to the **App Service logs** page from the app's portal page.
- Set **Application Logging (Filesystem)** to **On**.
- Choose the log **Level**. This setting only applies to Azure log streaming.

Navigate to the **Log Stream** page to view logs. The logged messages are logged with the `ILogger` interface.

## Azure Application Insights

The [Microsoft.Extensions.Logging.ApplicationInsights](#) provider package writes logs to [Azure Application Insights](#). Application Insights is a service that monitors a web app and provides tools for querying and analyzing the telemetry data. If you use this provider, you can query and analyze your logs by using the Application Insights tools.

The logging provider is included as a dependency of [Microsoft.ApplicationInsights.AspNetCore](#), which is the package that provides all available telemetry for ASP.NET Core. If you use this package, you don't have to install the provider package.

The [Microsoft.ApplicationInsights.Web](#) package is for ASP.NET 4.x, not ASP.NET Core.

For more information, see the following resources:

- [Application Insights overview](#)
- [Application Insights for ASP.NET Core applications](#): Start here if you want to implement the full range of Application Insights telemetry along with logging.
- [ApplicationInsightsLoggerProvider for .NET Core ILogger logs](#): Start here if you want to implement the logging provider without the rest of Application Insights telemetry.
- [Application Insights logging adapters](#)
- [Install, configure, and initialize the Application Insights SDK](#) interactive tutorial.

## Third-party logging providers



Third-party logging frameworks that work with ASP.NET Core:

- [elmah.io](#) [\(GitHub repo\)](#)
- [Gelf](#) [\(GitHub repo\)](#)
- [JSNLog](#) [\(GitHub repo\)](#)
- [KissLog.net](#) [\(GitHub repo\)](#)
- [Log4Net](#) [\(GitHub repo\)](#)
- [NLog](#) [\(GitHub repo\)](#)
- [PLogger](#) [\(GitHub repo\)](#)
- [Sentry](#) [\(GitHub repo\)](#)
- [Serilog](#) [\(GitHub repo\)](#)
- [Stackdriver](#) [\(Github repo\)](#)

Some third-party frameworks can perform [semantic logging](#), also known as [structured logging](#).

Using a third-party framework is similar to using one of the built-in providers:

1. Add a NuGet package to your project.
2. Call an `ILoggerFactory` extension method provided by the logging framework.

For more information, see each provider's documentation. Third-party logging providers aren't supported by Microsoft.

## No asynchronous logger methods

Logging should be so fast that it isn't worth the performance cost of asynchronous code. If a logging data store is slow, don't write to it directly. Consider writing the log messages to a fast store initially, then moving them to the slow store later. For example, when logging to SQL Server, don't do so directly in a `Log` method, since the `Log` methods are synchronous. Instead, synchronously add log messages to an in-memory queue and have a background worker pull the messages out of the queue to do the asynchronous work of pushing data to SQL Server. For more information, see [Guidance on how to log to a message queue for slow data stores \(dotnet/AspNetCore.Docs #11801\)](#).

## Change log levels in a running app

The Logging API doesn't include a scenario to change log levels while an app is running. However, some configuration providers are capable of reloading configuration, which takes immediate effect on logging configuration. For example, the [File Configuration Provider](#), reloads logging configuration by default. If configuration is changed in code

while an app is running, the app can call [IConfigurationRoot.Reload](#) to update the app's logging configuration.

## ILogger and ILoggerFactory

The [ILogger<TCategoryName>](#) and [ILoggerFactory](#) interfaces and implementations are included in the .NET Core SDK. They are also available in the following NuGet packages:

- The interfaces are in [Microsoft.Extensions.Logging.Abstractions](#) [↗](#).
- The default implementations are in [Microsoft.Extensions.Logging](#) [↗](#).

## Apply log filter rules in code

The preferred approach for setting log filter rules is by using [Configuration](#).

The following example shows how to register filter rules in code:

C#

```
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Logging.Debug;

var builder = WebApplication.CreateBuilder();
builder.Logging.AddFilter("System", LogLevel.Debug);
builder.Logging.AddFilter<DebugLoggerProvider>("Microsoft",
LogLevel.Information);
builder.Logging.AddFilter<ConsoleLoggerProvider>("Microsoft",
LogLevel.Trace);
```

`logging.AddFilter("System", LogLevel.Debug)` specifies the `System` category and log level `Debug`. The filter is applied to all providers because a specific provider was not configured.

`AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Information)` specifies:

- The `Debug` logging provider.
- Log level `Information` and higher.
- All categories starting with `"Microsoft"`.

## Automatically log scope with `SpanId`, `TraceId`, `ParentId`, `Baggage`, and `Tags`.

The logging libraries implicitly create a scope object with `SpanId`, `TraceId`, `ParentId`, `Baggage`, and `Tags`. This behavior is configured via [ActivityTrackingOptions](#).

C#

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

builder.Logging.AddSimpleConsole(options =>
{
    options.IncludeScopes = true;
});

builder.Logging.Configure(options =>
{
    options.ActivityTrackingOptions = ActivityTrackingOptions.SpanId
                                   | ActivityTrackingOptions.TraceId
                                   | ActivityTrackingOptions.ParentId
                                   | ActivityTrackingOptions.Baggage
                                   | ActivityTrackingOptions.Tags;
});
var app = builder.Build();

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

app.Run();
```

If the `traceparent` http request header is set, the `ParentId` in the log scope shows the W3C `parent-id` from in-bound `traceparent` header and the `SpanId` in the log scope shows the updated `parent-id` for the next out-bound step/span. For more information, see [Mutating the traceparent Field](#).

## Create a custom logger

To create a custom logger, see [Implement a custom logging provider in .NET](#).

## Additional resources

- [Improving logging performance with source generators](#)
- [Behind \[LogProperties\] and the new telemetry logging source generator](#)
- [Microsoft.Extensions.Logging source on GitHub](#)
- [View or download sample code](#) (how to download).
- [High performance logging](#)
- Logging bugs should be created in the [dotnet/runtime](#) GitHub repository.
- [ASP.NET Core Blazor logging](#)

# HTTP logging in ASP.NET Core

Article • 06/17/2024

## 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](#).

HTTP logging is a middleware that logs information about incoming HTTP requests and HTTP responses. HTTP logging provides logs of:

- HTTP request information
- Common properties
- Headers
- Body
- HTTP response information

HTTP logging can:

- Log all requests and responses or only requests and responses that meet certain criteria.
- Select which parts of the request and response are logged.
- Allow you to redact sensitive information from the logs.

HTTP logging *can reduce the performance of an app*, especially when logging the request and response bodies. Consider the performance impact when selecting fields to log. Test the performance impact of the selected logging properties.

## Warning

HTTP logging can potentially log personally identifiable information (PII). Consider the risk and avoid logging sensitive information.

## Enable HTTP logging

HTTP logging is enabled by calling [AddHttpLogging](#) and [UseHttpLogging](#), as shown in the following example:

C#

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

builder.Services.AddHttpLogging(o => { });

var app = builder.Build();

app.UseHttpLogging();

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

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

app.Run();
```

The empty lambda in the preceding example of calling `AddHttpLogging` adds the middleware with the default configuration. By default, HTTP logging logs common properties such as path, status-code, and headers for requests and responses.

Add the following line to the `appsettings.Development.json` file at the `"LogLevel": {` level so the HTTP logs are displayed:

JSON

```
"Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware": "Information"
```

With the default configuration, a request and response is logged as a pair of messages similar to the following example:

Output

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Protocol: HTTP/2
      Method: GET
      Scheme: https
      PathBase:
      Path: /
      Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,
*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
      Host: localhost:52941
      User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/118.0.0.0 Safari/537.36
```

```
Edg/118.0.2088.61
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: [Redacted]
sec-ch-ua: [Redacted]
sec-ch-ua-mobile: [Redacted]
sec-ch-ua-platform: [Redacted]
sec-fetch-site: [Redacted]
sec-fetch-mode: [Redacted]
sec-fetch-user: [Redacted]
sec-fetch-dest: [Redacted]
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
Response:
StatusCode: 200
Content-Type: text/plain; charset=utf-8
Date: Tue, 24 Oct 2023 02:03:53 GMT
Server: Kestrel
```

## HTTP logging options

To configure global options for the HTTP logging middleware, call [AddHttpLogging](#) in `Program.cs`, using the lambda to configure [HttpLoggingOptions](#).

C#

```
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("sec-ch-ua");
    logging.ResponseHeaders.Add("MyResponseHeader");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
    logging.CombineLogs = true;
});

var app = builder.Build();

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

app.UseStaticFiles();

app.UseHttpLogging();
```

```

app.Use(async (context, next) =>
{
    context.Response.Headers["MyResponseHeader"] =
        new string[] { "My Response Header Value" };

    await next();
});

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

app.Run();

```

### ⓘ Note

In the preceding sample and following samples, `UseHttpLogging` is called after `UseStaticFiles`, so HTTP logging is not enabled for static files. To enable static file HTTP logging, call `UseHttpLogging` before `UseStaticFiles`.

## LoggingFields

`HttpLoggingOptions.LoggingFields` is an enum flag that configures specific parts of the request and response to log. `HttpLoggingOptions.LoggingFields` defaults to `RequestPropertiesAndHeaders` | `ResponsePropertiesAndHeaders`.

## RequestHeaders and ResponseHeaders

`RequestHeaders` and `ResponseHeaders` are sets of HTTP headers that are logged. Header values are only logged for header names that are in these collections. The following code adds `sec-ch-ua` to the `RequestHeaders`, so the value of the `sec-ch-ua` header is logged. And it adds `MyResponseHeader` to the `ResponseHeaders`, so the value of the `MyResponseHeader` header is logged. If these lines are removed, the values of these headers are [Redacted].

C#

```

using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("sec-ch-ua");
    logging.ResponseHeaders.Add("MyResponseHeader");
});

```

```

        logging.MediaTypeOptions.AddText("application/javascript");
        logging.RequestBodyLogLimit = 4096;
        logging.ResponseBodyLogLimit = 4096;
        logging.CombineLogs = true;
    });

    var app = builder.Build();

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

    app.UseStaticFiles();

    app.UseHttpLogging();

    app.Use(async (context, next) =>
    {
        context.Response.Headers["MyResponseHeader"] =
            new string[] { "My Response Header Value" };

        await next();
    });

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

    app.Run();

```

## MediaTypeOptions

[MediaTypeOptions](#) provides configuration for selecting which encoding to use for a specific media type.

C#

```

using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("sec-ch-ua");
    logging.ResponseHeaders.Add("MyResponseHeader");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
    logging.CombineLogs = true;
});

var app = builder.Build();

```



```

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

app.UseStaticFiles();

app.UseHttpLogging();

app.Use(async (context, next) =>
{
    context.Response.Headers["MyResponseHeader"] =
        new string[] { "My Response Header Value" };

    await next();
});

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

app.Run();

```

This approach can also be used to enable logging for data that isn't logged by default (for example, form data, which might have a media type such as `application/x-www-form-urlencoded` or `multipart/form-data`).

## MediaTypeOptions methods

- [AddText](#)
- [AddBinary](#)
- [Clear](#)

## RequestBodyLogLimit and ResponseBodyLogLimit

- [RequestBodyLogLimit](#)
- [ResponseBodyLogLimit](#)

C#

```

using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("sec-ch-ua");
    logging.ResponseHeaders.Add("MyResponseHeader");
});

```

```

        logging.MediaTypeOptions.AddText("application/javascript");
        logging.RequestBodyLogLimit = 4096;
        logging.ResponseBodyLogLimit = 4096;
        logging.CombineLogs = true;
    });

    var app = builder.Build();

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

    app.UseStaticFiles();

    app.UseHttpLogging();

    app.Use(async (context, next) =>
    {
        context.Response.Headers["MyResponseHeader"] =
            new string[] { "My Response Header Value" };

        await next();
    });

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

    app.Run();

```

## CombineLogs

Setting `CombineLogs` to `true` configures the middleware to consolidate all of its enabled logs for a request and response into one log at the end. This includes the request, request body, response, response body, and duration.

C#

```

using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("sec-ch-ua");
    logging.ResponseHeaders.Add("MyResponseHeader");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
    logging.CombineLogs = true;
});

```

```

var app = builder.Build();

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

app.UseStaticFiles();

app.UseHttpLogging();

app.Use(async (context, next) =>
{
    context.Response.Headers["MyResponseHeader"] =
        new string[] { "My Response Header Value" };

    await next();
});

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

app.Run();

```

## Endpoint-specific configuration

For endpoint-specific configuration in minimal API apps, a [WithHttpLogging](#) extension method is available. The following example shows how to configure HTTP logging for one endpoint:

C#

```

app.MapGet("/response", () => "Hello World! (logging response)")
    .WithHttpLogging(HttpLoggingFields.ResponsePropertiesAndHeaders);

```

For endpoint-specific configuration in apps that use controllers, the [\[HttpLogging\]](#) attribute is available. The attribute can also be used in minimal API apps, as shown in the following example:

C#

```

app.MapGet("/duration", [HttpLogging(loggingFields:
    HttpLoggingFields.Duration)]
    () => "Hello World! (logging duration)");

```

## IHttpLoggingInterceptor

[IHttpLoggingInterceptor](#) is the interface for a service that can be implemented to handle per-request and per-response callbacks for customizing what details get logged. Any endpoint-specific log settings are applied first and can then be overridden in these callbacks. An implementation can:

- Inspect a request or response.
- Enable or disable any [HttpLoggingFields](#).
- Adjust how much of the request or response body is logged.
- Add custom fields to the logs.

Register an `IHttpLoggingInterceptor` implementation by calling [AddHttpLoggingInterceptor<T>](#) in `Program.cs`. If multiple `IHttpLoggingInterceptor` instances are registered, they're run in the order registered.

The following example shows how to register an `IHttpLoggingInterceptor` implementation:

C#

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

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.Duration;
});
builder.Services.AddHttpLoggingInterceptor<SampleHttpLoggingInterceptor>();
```

The following example is an `IHttpLoggingInterceptor` implementation that:

- Inspects the request method and disables logging for POST requests.
- For non-POST requests:
  - Redacts request path, request headers, and response headers.
  - Adds custom fields and field values to the request and response logs.

C#

```
using Microsoft.AspNetCore.HttpLogging;

namespace HttpLoggingSample;

internal sealed class SampleHttpLoggingInterceptor : IHttpLoggingInterceptor
{
    public ValueTask OnRequestAsync(HttpLoggingInterceptorContext
logContext)
    {
        if (logContext.HttpContext.Request.Method == "POST")
        {
```

```

        // Don't log anything if the request is a POST.
        logContext.LoggingFields = HttpLoggingFields.None;
    }

    // Don't enrich if we're not going to log any part of the request.
    if (!logContext.IsAnyEnabled(HttpLoggingFields.Request))
    {
        return default;
    }

    if (logContext.TryDisable(HttpLoggingFields.RequestPath))
    {
        RedactPath(logContext);
    }

    if (logContext.TryDisable(HttpLoggingFields.RequestHeaders))
    {
        RedactRequestHeaders(logContext);
    }

    EnrichRequest(logContext);

    return default;
}

public ValueTask OnResponseAsync(HttpLoggingInterceptorContext
logContext)
{
    // Don't enrich if we're not going to log any part of the response
    if (!logContext.IsAnyEnabled(HttpLoggingFields.Response))
    {
        return default;
    }

    if (logContext.TryDisable(HttpLoggingFields.ResponseHeaders))
    {
        RedactResponseHeaders(logContext);
    }

    EnrichResponse(logContext);

    return default;
}

private void RedactPath(HttpLoggingInterceptorContext logContext)
{
    logContext.AddParameter(nameof(logContext.HttpContext.Request.Path),
"RedactedPath");
}

private void RedactRequestHeaders(HttpLoggingInterceptorContext
logContext)
{
    foreach (var header in logContext.HttpContext.Request.Headers)
    {

```

```

        logContext.AddParameter(header.Key, "RedactedHeader");
    }
}

private void EnrichRequest(HttpLoggingInterceptorContext logContext)
{
    logContext.AddParameter("RequestEnrichment", "Stuff");
}

private void RedactResponseHeaders(HttpLoggingInterceptorContext logContext)
{
    foreach (var header in logContext.HttpContext.Response.Headers)
    {
        logContext.AddParameter(header.Key, "RedactedHeader");
    }
}

private void EnrichResponse(HttpLoggingInterceptorContext logContext)
{
    logContext.AddParameter("ResponseEnrichment", "Stuff");
}
}

```

With this interceptor, a POST request doesn't generate any logs even if HTTP logging is configured to log `HttpLoggingFields.All`. A GET request generates logs similar to the following example:

#### Output

```

info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
  Request:
  Path: RedactedPath
  Accept: RedactedHeader
  Host: RedactedHeader
  User-Agent: RedactedHeader
  Accept-Encoding: RedactedHeader
  Accept-Language: RedactedHeader
  Upgrade-Insecure-Requests: RedactedHeader
  sec-ch-ua: RedactedHeader
  sec-ch-ua-mobile: RedactedHeader
  sec-ch-ua-platform: RedactedHeader
  sec-fetch-site: RedactedHeader
  sec-fetch-mode: RedactedHeader
  sec-fetch-user: RedactedHeader
  sec-fetch-dest: RedactedHeader
  RequestEnrichment: Stuff
  Protocol: HTTP/2
  Method: GET
  Scheme: https
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]

```

```
Response:
Content-Type: RedactedHeader
MyResponseHeader: RedactedHeader
ResponseEnrichment: Stuff
StatusCode: 200
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[4]
      ResponseBody: Hello World!
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[8]
      Duration: 2.2778ms
```

## Logging configuration order of precedence

The following list shows the order of precedence for logging configuration:

1. Global configuration from [HttpLoggingOptions](#), set by calling [AddHttpLogging](#).
2. Endpoint-specific configuration from the [\[HttpLogging\]](#) attribute or the [WithHttpLogging](#) extension method overrides global configuration.
3. [IHttpLoggingInterceptor](#) is called with the results and can further modify the configuration per request.

# W3CLogger 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](#).

W3CLogger is a middleware that writes log files in the [W3C standard format](#). The logs contain information about HTTP requests and HTTP responses. W3CLogger provides logs of:

- HTTP request information
- Common properties
- Headers
- HTTP response information
- Metadata about the request/response pair (date/time started, time taken)

W3CLogger is valuable in several scenarios to:

- Record information about incoming requests and responses.
- Filter which parts of the request and response are logged.
- Filter which headers to log.

W3CLogger *can reduce the performance of an app*. Consider the performance impact when selecting fields to log - the performance reduction will increase as you log more properties. Test the performance impact of the selected logging properties.

## Warning

W3CLogger can potentially log personally identifiable information (PII). Consider the risk and avoid logging sensitive information. By default, fields that could contain PII aren't logged.

## Enable W3CLogger

Enable W3CLogger with [UseW3CLogging](#), which adds the W3CLogger middleware:



C#

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

app.UseW3CLogging();

app.UseRouting();
```

By default, W3CLogger logs common properties such as path, status-code, date, time, and protocol. All information about a single request/response pair is written to the same line.

```
#Version: 1.0
#Start-Date: 2021-09-29 22:18:28
#Fields: date time c-ip s-computername s-ip s-port cs-method cs-uri-stem cs-
uri-query sc-status time-taken cs-version cs-host cs(User-Agent) cs(Referer)
2021-09-29 22:18:28 ::1 DESKTOP-LH3TLTA ::1 5000 GET / - 200 59.9171
HTTP/1.1 localhost:5000 Mozilla/5.0+
(Windows+NT+10.0;+WOW64)+AppleWebKit/537.36+
(KHTML,+like+Gecko)+Chrome/93.0.4577.82+Safari/537.36 -
2021-09-29 22:18:28 ::1 DESKTOP-LH3TLTA ::1 5000 GET / - 200 0.1802 HTTP/1.1
localhost:5000 Mozilla/5.0+(Windows+NT+10.0;+WOW64)+AppleWebKit/537.36+
(KHTML,+like+Gecko)+Chrome/93.0.4577.82+Safari/537.36 -
2021-09-29 22:18:30 ::1 DESKTOP-LH3TLTA ::1 5000 GET / - 200 0.0966 HTTP/1.1
localhost:5000 Mozilla/5.0+(Windows+NT+10.0;+WOW64)+AppleWebKit/537.36+
(KHTML,+like+Gecko)+Chrome/93.0.4577.82+Safari/537.36 -
```

## W3CLogger options

To configure the W3CLogger middleware, call [AddW3CLogging](#) in `Program.cs`:

C#

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

builder.Services.AddW3CLogging(logging =>
{
    // Log all W3C fields
    logging.LoggingFields = W3CLoggingFields.All;

    logging.AdditionalRequestHeaders.Add("x-forwarded-for");
    logging.AdditionalRequestHeaders.Add("x-client-ssl-protocol");
    logging.FileSizeLimit = 5 * 1024 * 1024;
    logging.RetainedFileCountLimit = 2;
    logging.FileName = "MyLogFile";
    logging.LogDirectory = @"C:\logs";
});
```

```
logging.FlushInterval = TimeSpan.FromSeconds(2);  
});
```

## LoggingFields

[W3CLoggerOptions.LoggingFields](#) is a bit flag enumeration that configures specific parts of the request and response to log, and other information about the connection.

`LoggingFields` defaults to include all possible fields except `UserName` and `Cookie`. For a complete list of available fields, see [W3CLoggingFields](#).

# Health checks in ASP.NET Core

Article • 07/23/2024

By [Glenn Condrón](#) and [Juergen Gutsch](#)

## 📘 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 offers Health Checks Middleware and libraries for reporting the health of app infrastructure components.

Health checks are exposed by an app as HTTP endpoints. Health check endpoints can be configured for various real-time monitoring scenarios:

- Health probes can be used by container orchestrators and load balancers to check an app's status. For example, a container orchestrator may respond to a failing health check by halting a rolling deployment or restarting a container. A load balancer might react to an unhealthy app by routing traffic away from the failing instance to a healthy instance.
- Use of memory, disk, and other physical server resources can be monitored for healthy status.
- Health checks can test an app's dependencies, such as databases and external service endpoints, to confirm availability and normal functioning.

Health checks are typically used with an external monitoring service or container orchestrator to check the status of an app. Before adding health checks to an app, decide on which monitoring system to use. The monitoring system dictates what types of health checks to create and how to configure their endpoints.

## Basic health probe

For many apps, a basic health probe configuration that reports the app's availability to process requests (*liveness*) is sufficient to discover the status of the app.

The basic configuration registers health check services and calls the Health Checks Middleware to respond at a URL endpoint with a health response. By default, no specific

health checks are registered to test any particular dependency or subsystem. The app is considered healthy if it can respond at the health endpoint URL. The default response writer writes [HealthStatus](#) as a plaintext response to the client. The `HealthStatus` is [HealthStatus.Healthy](#), [HealthStatus.Degraded](#), or [HealthStatus.Unhealthy](#).

Register health check services with [AddHealthChecks](#) in `Program.cs`. Create a health check endpoint by calling [MapHealthChecks](#).

The following example creates a health check endpoint at `/healthz`:

C#

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

builder.Services.AddHealthChecks();

var app = builder.Build();

app.MapHealthChecks("/healthz");

app.Run();
```

## Docker HEALTHCHECK

[Docker](#) offers a built-in `HEALTHCHECK` directive that can be used to check the status of an app that uses the basic health check configuration:

Dockerfile

```
HEALTHCHECK CMD curl --fail http://localhost:5000/healthz || exit
```

The preceding example uses `curl` to make an HTTP request to the health check endpoint at `/healthz`. `curl` isn't included in the .NET Linux container images, but it can be added by installing the required package in the Dockerfile. Containers that use images based on Alpine Linux can use the included `wget` in place of `curl`.

## Create health checks

Health checks are created by implementing the [IHealthCheck](#) interface. The [CheckHealthAsync](#) method returns a [HealthCheckResult](#) that indicates the health as `Healthy`, `Degraded`, or `Unhealthy`. The result is written as a plaintext response with a configurable status code. Configuration is described in the [Health check options](#) section. [HealthCheckResult](#) can also return optional key-value pairs.

The following example demonstrates the layout of a health check:

C#

```
public class SampleHealthCheck : IHealthCheck
{
    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken =
default)
    {
        var isHealthy = true;

        // ...

        if (isHealthy)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("A healthy result.));
        }

        return Task.FromResult(
            new HealthCheckResult(
                context.Registration.FailureStatus, "An unhealthy
result.));
    }
}
```

The health check's logic is placed in the `CheckHealthAsync` method. The preceding example sets a dummy variable, `isHealthy`, to `true`. If the value of `isHealthy` is set to `false`, the `HealthCheckRegistration.FailureStatus` status is returned.

If `CheckHealthAsync` throws an exception during the check, a new `HealthReportEntry` is returned with its `HealthReportEntry.Status` set to the `FailureStatus`. This status is defined by `AddCheck` (see the [Register health check services](#) section) and includes the `inner exception` that caused the check failure. The `Description` is set to the exception's message.

## Register health check services

To register a health check service, call `AddCheck` in `Program.cs`:

C#

```
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheck>("Sample");
```

The [AddCheck](#) overload shown in the following example sets the failure status ([HealthStatus](#)) to report when the health check reports a failure. If the failure status is set to `null` (default), [HealthStatus.Unhealthy](#) is reported. This overload is a useful scenario for library authors, where the failure status indicated by the library is enforced by the app when a health check failure occurs if the health check implementation honors the setting.

*Tags* can be used to filter health checks. Tags are described in the [Filter health checks](#) section.

C#

```
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheck>(
        "Sample",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "sample" });
```

[AddCheck](#) can also execute a lambda function. In the following example, the health check always returns a healthy result:

C#

```
builder.Services.AddHealthChecks()
    .AddCheck("Sample", () => HealthCheckResult.Healthy("A healthy
result."));
```

Call [AddTypeActivatedCheck](#) to pass arguments to a health check implementation. In the following example, a type-activated health check accepts an integer and a string in its constructor:

C#

```
public class SampleHealthCheckWithArgs : IHealthCheck
{
    private readonly int _arg1;
    private readonly string _arg2;

    public SampleHealthCheckWithArgs(int arg1, string arg2)
        => (_arg1, _arg2) = (arg1, arg2);

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken =
default)
    {
        // ...
    }
}
```

```
        return Task.FromResult(HealthCheckResult.Healthy("A healthy  
result."));  
    }  
}
```

To register the preceding health check, call `AddTypeActivatedCheck` with the integer and string passed as arguments:

C#

```
builder.Services.AddHealthChecks()  
    .AddTypeActivatedCheck<SampleHealthCheckWithArgs>(  
        "Sample",  
        failureStatus: HealthStatus.Degraded,  
        tags: new[] { "sample" },  
        args: new object[] { 1, "Arg" }));
```

## Use Health Checks Routing

In `Program.cs`, call `MapHealthChecks` on the endpoint builder with the endpoint URL or relative path:

C#

```
app.MapHealthChecks("/healthz");
```

## Require host

Call `RequireHost` to specify one or more permitted hosts for the health check endpoint. Hosts should be Unicode rather than punycode and may include a port. If a collection isn't supplied, any host is accepted:

C#


```
app.MapHealthChecks("/healthz")  
    .RequireHost("www.contoso.com:5001");
```

To restrict the health check endpoint to respond only on a specific port, specify a port in the call to `RequireHost`. This approach is typically used in a container environment to expose a port for monitoring services:

C#

```
app.MapHealthChecks("/healthz")
    .RequireHost("*:5001");
```

### Warning

API that relies on the [Host header](#) , such as [HttpRequest.Host](#) and [RequireHost](#), are subject to potential spoofing by clients.

To prevent host and port spoofing, use one of the following approaches:

- Use [HttpContext.Connection](#) ([ConnectionInfo.LocalPort](#)) where the ports are checked.
- Employ [Host filtering](#).

To prevent unauthorized clients from spoofing the port, call [RequireAuthorization](#):

C#

```
app.MapHealthChecks("/healthz")
    .RequireHost("*:5001")
    .RequireAuthorization();
```

For more information, see [Host matching in routes with RequireHost](#).


## Require authorization

Call [RequireAuthorization](#) to run Authorization Middleware on the health check request endpoint. A `RequireAuthorization` overload accepts one or more authorization policies. If a policy isn't provided, the default authorization policy is used:

C#

```
app.MapHealthChecks("/healthz")
    .RequireAuthorization();
```

## Enable Cross-Origin Requests (CORS)

Although running health checks manually from a browser isn't a common scenario, CORS Middleware can be enabled by calling [RequireCors](#)  on the health checks endpoints. The [RequireCors](#) overload accepts a CORS policy builder delegate



(`CorsPolicyBuilder`) or a policy name. For more information, see [Enable Cross-Origin Requests \(CORS\) in ASP.NET Core](#).

## Health check options

[HealthCheckOptions](#) provide an opportunity to customize health check behavior:

- [Filter health checks](#)
- [Customize the HTTP status code](#)
- [Suppress cache headers](#)
- [Customize output](#)

### Filter health checks

By default, the Health Checks Middleware runs all registered health checks. To run a subset of health checks, provide a function that returns a boolean to the [Predicate](#) option.

The following example filters the health checks so that only those tagged with `sample` run:

C#

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    Predicate = healthCheck => healthCheck.Tags.Contains("sample")
});
```

### Customize the HTTP status code

Use [ResultStatusCodes](#) to customize the mapping of health status to HTTP status codes. The following [StatusCodes](#) assignments are the default values used by the middleware. Change the status code values to meet your requirements:

C#

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    ResultStatusCodes =
    {
        [HealthStatus.Healthy] = StatusCodes.Status200OK,
        [HealthStatus.Degraded] = StatusCodes.Status200OK,
        [HealthStatus.Unhealthy] = StatusCodes.Status503ServiceUnavailable
    }
});
```

```
    }  
});
```

## Suppress cache headers

[AllowCachingResponses](#) controls whether the Health Checks Middleware adds HTTP headers to a probe response to prevent response caching. If the value is `false` (default), the middleware sets or overrides the `Cache-Control`, `Expires`, and `Pragma` headers to prevent response caching. If the value is `true`, the middleware doesn't modify the cache headers of the response:

C#

```
app.MapHealthChecks("/healthz", new HealthCheckOptions  
{  
    AllowCachingResponses = true  
});
```

## Customize output

To customize the output of a health checks report, set the [HealthCheckOptions.ResponseWriter](#) property to a delegate that writes the response:

C#

```
app.MapHealthChecks("/healthz", new HealthCheckOptions  
{  
    ResponseWriter = WriteResponse  
});
```

The default delegate writes a minimal plaintext response with the string value of [HealthReport.Status](#). The following custom delegate outputs a custom JSON response using [System.Text.Json](#):

C#

```
private static Task WriteResponse(HttpContext context, HealthReport  
healthReport)  
{  
    context.Response.ContentType = "application/json; charset=utf-8";  
  
    var options = new JsonSerializerOptions { Indented = true };  
  
    using var memoryStream = new MemoryStream();  
    using (var jsonWriter = new Utf8JsonWriter(memoryStream, options))
```

```

{
    jsonWriter.WriteStartObject();
    jsonWriter.WriteString("status", healthReport.Status.ToString());
    jsonWriter.WriteStartObject("results");

    foreach (var healthReportEntry in healthReport.Entries)
    {
        jsonWriter.WriteStartObject(healthReportEntry.Key);
        jsonWriter.WriteString("status",
            healthReportEntry.Value.Status.ToString());
        jsonWriter.WriteString("description",
            healthReportEntry.Value.Description);
        jsonWriter.WriteStartObject("data");

        foreach (var item in healthReportEntry.Value.Data)
        {
            jsonWriter.WritePropertyName(item.Key);

            JsonSerializer.Serialize(jsonWriter, item.Value,
                item.Value?.GetType() ?? typeof(object));
        }

        jsonWriter.WriteEndObject();
        jsonWriter.WriteEndObject();
    }

    jsonWriter.WriteEndObject();
    jsonWriter.WriteEndObject();
}

return context.Response.WriteAsync(
    Encoding.UTF8.GetString(memoryStream.ToArray()));
}

```

The health checks API doesn't provide built-in support for complex JSON return formats because the format is specific to your choice of monitoring system. Customize the response in the preceding examples as needed. For more information on JSON serialization with `System.Text.Json`, see [How to serialize and deserialize JSON in .NET](#).

## Database probe

A health check can specify a database query to run as a boolean test to indicate if the database is responding normally.

[AspNetCore.Diagnostics.HealthChecks](#), a health check library for ASP.NET Core apps, includes a health check that runs against a SQL Server database.

`AspNetCore.Diagnostics.HealthChecks` executes a `SELECT 1` query against the database to confirm the connection to the database is healthy.

### ⚠ Warning

When checking a database connection with a query, choose a query that returns quickly. The query approach runs the risk of overloading the database and degrading its performance. In most cases, running a test query isn't necessary. Merely making a successful connection to the database is sufficient. If you find it necessary to run a query, choose a simple SELECT query, such as `SELECT 1`.

To use this SQL Server health check, include a package reference to the [AspNetCore.HealthChecks.SqlServer](#) NuGet package. The following example registers the SQL Server health check:

C#

```
var conStr = builder.Configuration.GetConnectionString("DefaultConnection");
if (string.IsNullOrEmpty(conStr))
{
    throw new InvalidOperationException(
        "Could not find a connection string named 'DefaultConnection'.");
}
builder.Services.AddHealthChecks()
    .AddSqlServer(conStr);
```

### ⓘ Note

[AspNetCore.Diagnostics.HealthChecks](#) isn't maintained or supported by Microsoft.

## Entity Framework Core DbContext probe

The `DbContext` check confirms that the app can communicate with the database configured for an EF Core `DbContext`. The `DbContext` check is supported in apps that:

- Use [Entity Framework \(EF\) Core](#).
- Include a package reference to the [Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore](#) NuGet package.

`AddDbContextCheck` registers a health check for a `DbContext`. The `DbContext` is supplied to the method as the `TContext`. An overload is available to configure the failure

status, tags, and a custom test query.

By default:

- The `DbContextHealthCheck` calls EF Core's `CanConnectAsync` method. You can customize what operation is run when checking health using `AddDbContextCheck` method overloads.
- The name of the health check is the name of the `TContext` type.

The following example registers a `DbContext` and an associated `DbContextHealthCheck`:

C#

```
builder.Services.AddDbContext<SampleDbContext>(options =>
    options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddHealthChecks()
    .AddDbContextCheck<SampleDbContext>();
```

## Separate readiness and liveness probes

In some hosting scenarios, a pair of health checks is used to distinguish two app states:

- *Readiness* indicates if the app is running normally but isn't ready to receive requests.
- *Liveness* indicates if an app has crashed and must be restarted.

Consider the following example: An app must download a large configuration file before it's ready to process requests. We don't want the app to be restarted if the initial download fails because the app can retry downloading the file several times. We use a *liveness probe* to describe the liveness of the process, no other checks are run. We also want to prevent requests from being sent to the app before the configuration file download has succeeded. We use a *readiness probe* to indicate a "not ready" state until the download succeeds and the app is ready to receive requests.

The following background task simulates a startup process that takes roughly 15 seconds. Once it completes, the task sets the `StartupHealthCheck.StartupCompleted` property to true:

C#

```
public class StartupBackgroundService : BackgroundService
{
    private readonly StartupHealthCheck _healthCheck;
```

```

public StartupBackgroundService(StartupHealthCheck healthCheck)
    => _healthCheck = healthCheck;

protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
{
    // Simulate the effect of a long-running task.
    await Task.Delay(TimeSpan.FromSeconds(15), stoppingToken);

    _healthCheck.StartupCompleted = true;
}
}

```

The `StartupHealthCheck` reports the completion of the long-running startup task and exposes the `StartupCompleted` property that gets set by the background service:

C#

```

public class StartupHealthCheck : IHealthCheck
{
    private volatile bool _isReady;

    public bool StartupCompleted
    {
        get => _isReady;
        set => _isReady = value;
    }

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken =
default)
    {
        if (StartupCompleted)
        {
            return Task.FromResult(HealthCheckResult.Healthy("The startup
task has completed."));
        }

        return Task.FromResult(HealthCheckResult.Unhealthy("That startup
task is still running."));
    }
}

```

The health check is registered with `AddCheck` in `Program.cs` along with the hosted service. Because the hosted service must set the property on the health check, the health check is also registered in the service container as a singleton:

C#

```
builder.Services.AddHostedService<StartupBackgroundService>();
builder.Services.AddSingleton<StartupHealthCheck>();

builder.Services.AddHealthChecks()
    .AddCheck<StartupHealthCheck>(
        "Startup",
        tags: new[] { "ready" });
```

To create two different health check endpoints, call `MapHealthChecks` twice:

C#

```
app.MapHealthChecks("/healthz/ready", new HealthCheckOptions
{
    Predicate = healthCheck => healthCheck.Tags.Contains("ready")
});

app.MapHealthChecks("/healthz/live", new HealthCheckOptions
{
    Predicate = _ => false
});
```

The preceding example creates the following health check endpoints:

- `/healthz/ready` for the readiness check. The readiness check filters health checks to those tagged with `ready`.
- `/healthz/live` for the liveness check. The liveness check filters out all health checks by returning `false` in the `HealthCheckOptions.Predicate` delegate. For more information on filtering health checks, see [Filter health checks](#) in this article.

Before the startup task completes, the `/healthz/ready` endpoint reports an `Unhealthy` status. Once the startup task completes, this endpoint reports a `Healthy` status. The `/healthz/live` endpoint excludes all checks and reports a `Healthy` status for all calls.

## Kubernetes example

Using separate readiness and liveness checks is useful in an environment such as [Kubernetes](#). In Kubernetes, an app might be required to run time-consuming startup work before accepting requests, such as a test of the underlying database availability. Using separate checks allows the orchestrator to distinguish whether the app is functioning but not yet ready or if the app has failed to start. For more information on readiness and liveness probes in Kubernetes, see [Configure Liveness and Readiness Probes](#) in the Kubernetes documentation.

The following example demonstrates a Kubernetes readiness probe configuration:

YAML

```
spec:
  template:
    spec:
      readinessProbe:
        # an http probe
        httpGet:
          path: /healthz/ready
          port: 80
        # length of time to wait for a pod to initialize
        # after pod startup, before applying health checking
        initialDelaySeconds: 30
        timeoutSeconds: 1
      ports:
        - containerPort: 80
```

## Distribute a health check library

To distribute a health check as a library:

1. Write a health check that implements the [IHealthCheck](#) interface as a standalone class. The class can rely on [dependency injection \(DI\)](#), type activation, and [named options](#) to access configuration data.
2. Write an extension method with parameters that the consuming app calls in its `Program.cs` method. Consider the following example health check, which accepts `arg1` and `arg2` as constructor parameters:

C#

```
public SampleHealthCheckWithArgs(int arg1, string arg2)
    => (_arg1, _arg2) = (arg1, arg2);
```

The preceding signature indicates that the health check requires custom data to process the health check probe logic. The data is provided to the delegate used to create the health check instance when the health check is registered with an extension method. In the following example, the caller specifies:

- `arg1`: An integer data point for the health check.
- `arg2`: A string argument for the health check.
- `name`: An optional health check name. If `null`, a default value is used.



- `failureStatus`: An optional [HealthStatus](#), which is reported for a failure status. If `null`, [HealthStatus.Unhealthy](#) is used.
- `tags`: An optional `IEnumerable<string>` collection of tags.

C#

```
public static class SampleHealthCheckBuilderExtensions
{
    private const string DefaultName = "Sample";

    public static IHealthChecksBuilder AddSampleHealthCheck(
        this IHealthChecksBuilder healthChecksBuilder,
        int arg1,
        string arg2,
        string? name = null,
        HealthStatus? failureStatus = null,
        IEnumerable<string>? tags = default)
    {
        return healthChecksBuilder.Add(
            new HealthCheckRegistration(
                name ?? DefaultName,
                _ => new SampleHealthCheckWithArgs(arg1, arg2),
                failureStatus,
                tags));
    }
}
```

## Health Check Publisher

When an [IHealthCheckPublisher](#) is added to the service container, the health check system periodically executes your health checks and calls [PublishAsync](#) with the result. This process is useful in a push-based health monitoring system scenario that expects each process to call the monitoring system periodically to determine health.

[HealthCheckPublisherOptions](#) allow you to set the:

- **Delay**: The initial delay applied after the app starts before executing [IHealthCheckPublisher](#) instances. The delay is applied once at startup and doesn't apply to later iterations. The default value is five seconds.
- **Period**: The period of [IHealthCheckPublisher](#) execution. The default value is 30 seconds.
- **Predicate**: If **Predicate** is `null` (default), the health check publisher service runs all registered health checks. To run a subset of health checks, provide a function that filters the set of checks. The predicate is evaluated each period.
- **Timeout**: The timeout for executing the health checks for all [IHealthCheckPublisher](#) instances. Use [InfiniteTimeSpan](#) to execute without a timeout. The default value is

30 seconds.

The following example demonstrates the layout of a health publisher:

C#

```
public class SampleHealthCheckPublisher : IHealthCheckPublisher
{
    public Task PublishAsync(HealthReport report, CancellationToken
cancellationToken)
    {
        if (report.Status == HealthStatus.Healthy)
        {
            // ...
        }
        else
        {
            // ...
        }

        return Task.CompletedTask;
    }
}
```

The [HealthCheckPublisherOptions](#) class provides properties for configuring the behavior of the health check publisher.

The following example registers a health check publisher as a singleton and configures [HealthCheckPublisherOptions](#):

C#

```
builder.Services.Configure<HealthCheckPublisherOptions>(options =>
{
    options.Delay = TimeSpan.FromSeconds(2);
    options.Predicate = healthCheck => healthCheck.Tags.Contains("sample");
});

builder.Services.AddSingleton<IHealthCheckPublisher,
SampleHealthCheckPublisher>();
```

[AspNetCore.Diagnostics.HealthChecks](#) [↗](#):

- Includes publishers for several systems, including [Application Insights](#).
- Is **not** maintained or supported by Microsoft.

## Individual Healthchecks

[Delay and Period](#) can be set on each [HealthCheckRegistration](#) individually. This is useful when you want to run some health checks at a different rate than the period set in [HealthCheckPublisherOptions](#).

The following code sets the `Delay` and `Period` for the `SampleHealthCheck1`:

C#

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

builder.Services.AddHealthChecks()
    .Add(new HealthCheckRegistration(
        name: "SampleHealthCheck1",
        instance: new SampleHealthCheck(),
        failureStatus: null,
        tags: null,
        timeout: default)
    {
        Delay = TimeSpan.FromSeconds(40),
        Period = TimeSpan.FromSeconds(30)
    });

var app = builder.Build();

app.MapHealthChecks("/healthz");

app.Run();
```

## Dependency Injection and Health Checks

It's possible to use dependency injection to consume an instance of a specific `Type` inside a Health Check class. Dependency injection can be useful to inject options or a global configuration to a Health Check. Using dependency injection is ***not*** a common scenario to configure Health Checks. Usually, each Health Check is quite specific to the actual test and is configured using `IHealthChecksBuilder` extension methods.

The following example shows a sample Health Check that retrieves a configuration object via dependency injection:

C#

```
public class SampleHealthCheckWithDI : IHealthCheck
{
    private readonly SampleHealthCheckWithDiConfig _config;

    public SampleHealthCheckWithDI(SampleHealthCheckWithDiConfig config)
        => _config = config;
```

```

public Task<HealthCheckResult> CheckHealthAsync(
    HealthCheckContext context, CancellationToken cancellationToken =
default)
{
    var isHealthy = true;

    // use _config ...

    if (isHealthy)
    {
        return Task.FromResult(
            HealthCheckResult.Healthy("A healthy result.));
    }

    return Task.FromResult(
        new HealthCheckResult(
            context.Registration.FailureStatus, "An unhealthy
result.));
    }
}

```

The `SampleHealthCheckWithDiConfig` and the Health check needs to be added to the service container :

C#

```

builder.Services.AddSingleton<SampleHealthCheckWithDiConfig>(new
SampleHealthCheckWithDiConfig
{
    BaseUriToCheck = new Uri("https://sample.contoso.com/api/")
});
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheckWithDI>(
        "With Dependency Injection",
        tags: new[] { "inject" });

```

## UseHealthChecks vs. MapHealthChecks


There are two ways to make health checks accessible to callers:

- [UseHealthChecks](#) registers middleware for handling health checks requests in the middleware pipeline.
- [MapHealthChecks](#) registers a health checks endpoint. The endpoint is matched and executed along with other endpoints in the app.

The advantage of using `MapHealthChecks` over `UseHealthChecks` is the ability to use endpoint aware middleware, such as authorization, and to have greater fine-grained control over the matching policy. The primary advantage of using `UseHealthChecks` over

`MapHealthChecks` is controlling exactly where health checks runs in the middleware pipeline.


#### UseHealthChecks:

- Terminates the pipeline when a request matches the health check endpoint. [Short-circuiting](#) is often desirable because it avoids unnecessary work, such as logging and other middleware.
- Is primarily used for configuring the health check middleware in the pipeline.
- Can match any path on a port with a `null` or empty `PathString`. Allows performing a health check on any request made to the specified port.
- [Source code](#) 

#### MapHealthChecks allows:

- Terminating the pipeline when a request matches the health check endpoint, by calling [ShortCircuit](#). For example, 


```
app.MapHealthChecks("/healthz").ShortCircuit();
```

 For more information, see [Short-circuit middleware after routing](#).
- Mapping specific routes or endpoints for health checks.
- Customization of the URL or path where the health check endpoint is accessible.
- Mapping multiple health check endpoints with different routes or configurations. Multiple endpoint support:
  - Enables separate endpoints for different types of health checks or components.
  - Is used to differentiate between different aspects of the app's health or apply specific configurations to subsets of health checks.
- [Source code](#) 

## Additional resources

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

### Note

This article was partially created with the help of artificial intelligence. Before publishing, an author reviewed and revised the content as needed. See [Our principles for using AI-generated content in Microsoft Learn](#) .

# ASP.NET Core metrics

Article • 11/14/2024

Metrics are numerical measurements reported over time. They're typically used to monitor the health of an app and generate alerts. For example, a web service might track how many:

- Requests it received per second.
- Milliseconds it took to respond.
- Responses sent an error.

These metrics can be reported to a monitoring system at regular intervals. Dashboards can be setup to view metrics and alerts created to notify people of problems. If the web service is intended to respond to requests within 400 ms and starts responding in 600 ms, the monitoring system can notify the operations staff that the app response is slower than normal.

## Tip

See [ASP.NET Core metrics](#) for a comprehensive list of all instruments together with their attributes.

## Using metrics

There are two parts to using metrics in a .NET app:

- **Instrumentation:** Code in .NET libraries takes measurements and associates these measurements with a metric name. .NET and ASP.NET Core include many built-in metrics.
- **Collection:** A .NET app configures named metrics to be transmitted from the app for external storage and analysis. Some tools may perform configuration outside the app using configuration files or a UI tool.

Instrumented code can record numeric measurements, but the measurements need to be aggregated, transmitted, and stored to create useful metrics for monitoring. The process of aggregating, transmitting, and storing data is called collection. This tutorial shows several examples of collecting metrics:

- Populating metrics in [Grafana](#) with [OpenTelemetry](#) and [Prometheus](#).
- Viewing metrics in real time with [dotnet-counters](#)

Measurements can also be associated with key-value pairs called tags that allow data to be categorized for analysis. For more information, see [Multi-dimensional metrics](#).

## Create the starter app

Create a new ASP.NET Core app with the following command:

.NET CLI

```
dotnet new web -o WebMetric
cd WebMetric
dotnet add package OpenTelemetry.Exporter.Prometheus.AspNetCore --prerelease
dotnet add package OpenTelemetry.Extensions.Hosting
```

Replace the contents of `Program.cs` with the following code:

C#

```
using OpenTelemetry.Metrics;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddOpenTelemetry()
    .WithMetrics(builder =>
    {
        builder.AddPrometheusExporter();

        builder.AddMeter("Microsoft.AspNetCore.Hosting",
            "Microsoft.AspNetCore.Server.Kestrel");
        builder.AddView("http.server.request.duration",
            new ExplicitBucketHistogramConfiguration
            {
                Boundaries = new double[] { 0, 0.005, 0.01, 0.025, 0.05,
                    0.075, 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10 }
            });
    });
var app = builder.Build();

app.MapPrometheusScrapingEndpoint();

app.MapGet("/", () => "Hello OpenTelemetry! ticks:"
    + DateTime.Now.Ticks.ToString()[^3..]);

app.Run();
```

## View metrics with dotnet-counters

[dotnet-counters](#) is a command-line tool that can view live metrics for .NET Core apps on demand. It doesn't require setup, making it useful for ad-hoc investigations or verifying that metric instrumentation is working. It works with both [System.Diagnostics.Metrics](#) based APIs and [EventCounters](#).

If the [dotnet-counters](#) tool isn't installed, run the following command:

```
.NET CLI
```

```
dotnet tool update -g dotnet-counters
```

While the test app is running, launch [dotnet-counters](#). The following command shows an example of `dotnet-counters` monitoring all metrics from the [Microsoft.AspNetCore.Hosting](#) meter.

```
.NET CLI
```

```
dotnet-counters monitor -n WebMetric --counters Microsoft.AspNetCore.Hosting
```

Output similar to the following is displayed:

```
.NET CLI
```

```
Press p to pause, r to resume, q to quit.
```

```
Status: Running
```

```
[Microsoft.AspNetCore.Hosting]
```

```
http-server-current-requests
```

```
host=localhost,method=GET,port=5045,scheme=http 0
```

```
http-server-request-duration (s)
```

```
host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro
```

```
0.001
```

```
host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro
```

```
0.001
```

```
host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro
```

```
0.001
```

```
host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro 0
```

```
host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro 0
```

```
host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro 0
```

For more information, see [dotnet-counters](#).

## Enrich the ASP.NET Core request metric

ASP.NET Core has many built-in metrics. The `http.server.request.duration` metric:



- Records the duration of HTTP requests on the server.
- Captures request information in tags, such as the matched route and response status code.

The `http.server.request.duration` metric supports tag enrichment using [IHttpMetricsTagsFeature](#). Enrichment is when a library or app adds its own tags to a metric. This is useful if an app wants to add a custom categorization to dashboards or alerts built with metrics.

C#

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

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

app.Use(async (context, next) =>
{
    var tagsFeature = context.Features.Get<IHttpMetricsTagsFeature>();
    if (tagsFeature != null)
    {
        var source = context.Request.Query["utm_medium"].ToString() switch
        {
            "" => "none",
            "social" => "social",
            "email" => "email",
            "organic" => "organic",
            _ => "other"
        };
        tagsFeature.Tags.Add(new KeyValuePair<string, object?>("mkt_medium",
source));
    }

    await next.Invoke();
});

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

app.Run();
```

The proceeding example:

- Adds middleware to enrich the ASP.NET Core request metric.
- Gets the [IHttpMetricsTagsFeature](#) from the `HttpContext`. The feature is only present on the context if someone is listening to the metric. Verify `IHttpMetricsTagsFeature` is not `null` before using it.
- Adds a custom tag containing the request's marketing source to the [http.server.request.duration](#) metric.

- The tag has the name `mkt_medium` and a value based on the `utm_medium` [query string value](#). The `utm_medium` value is resolved to a known range of values.
- The tag allows requests to be categorized by marketing medium type, which could be useful when analyzing web app traffic.

#### ⓘ Note

Follow the [multi-dimensional metrics](#) best practices when enriching with custom tags. Tags that are too numerous or have an unbound range create many tag combinations, resulting in high dimensions. Collection tools have limits on supported dimensions for a counter and may filter results to prevent excessive memory use.

## Opt-out of HTTP metrics on certain endpoints and requests

Opting out of recording metrics is beneficial for endpoints frequently called by automated systems, such as health checks. Recording metrics for these requests is generally unnecessary. Unwanted telemetry uses resources to collect and store, and can distort results displayed in a telemetry dashboard.

HTTP requests to an endpoint can be excluded from metrics by adding metadata, with either the [DisableHttpMetrics](#) attribute or the [DisableHttpMetrics](#) method:

- Add the [DisableHttpMetrics](#) attribute to the Web API controller, SignalR hub or gRPC service.
- Call [DisableHttpMetrics](#) when mapping endpoints in app startup:

C#

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

var app = builder.Build();
app.MapHealthChecks("/healthz").DisableHttpMetrics();
app.Run();
```

Alternatively, the [IHttpMetricsTagsFeature.MetricsDisabled](#) property has been added for:

- Advanced scenarios where a request doesn't map to an endpoint.
- Dynamically disabling metrics collection for specific HTTP requests.

C#

```
// Middleware that conditionally opts-out HTTP requests.
app.Use(async (context, next) =>
{
    var metricsFeature = context.Features.Get<IHttpMetricsTagsFeature>();
    if (metricsFeature != null &&
        context.Request.Headers.ContainsKey("x-disable-metrics"))
    {
        metricsFeature.MetricsDisabled = true;
    }

    await next(context);
});
```

## Create custom metrics

Metrics are created using APIs in the [System.Diagnostics.Metrics](#) namespace. See [Create custom metrics](#) for information on creating custom metrics.

## Creating metrics in ASP.NET Core apps with `IMeterFactory`

We recommend creating [Meter](#) instances in ASP.NET Core apps with [IMeterFactory](#).

ASP.NET Core registers [IMeterFactory](#) in dependency injection (DI) by default. The meter factory integrates metrics with DI, making isolating and collecting metrics easy.

`IMeterFactory` is especially useful for testing. It allows for multiple tests to run side-by-side and only collecting metrics values that are recorded in a test.

To use `IMeterFactory` in an app, create a type that uses `IMeterFactory` to create the app's custom metrics:

C#

```
public class ContosoMetrics
{
    private readonly Counter<int> _productSoldCounter;

    public ContosoMetrics(IMeterFactory meterFactory)
    {
        var meter = meterFactory.Create("Contoso.Web");
        _productSoldCounter = meter.CreateCounter<int>
("contoso.product.sold");
    }

    public void ProductSold(string productName, int quantity)
```

```

    {
        _productSoldCounter.Add(quantity,
            new KeyValuePair<string, object?>("contoso.product.name",
productName));
    }
}

```

Register the metrics type with DI in `Program.cs`:

C#

```

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<ContosoMetrics>();

```

Inject the metrics type and record values where needed. Because the metrics type is registered in DI it can be use with MVC controllers, minimal APIs, or any other type that is created by DI:

C#

```

app.MapPost("/complete-sale", (SaleModel model, ContosoMetrics metrics) =>
{
    // ... business logic such as saving the sale to a database ...

    metrics.ProductSold(model.ProductName, model.QuantitySold);
});

```

To monitor the "Contoso.Web" meter, use the following [dotnet-counters](#) command.

.NET CLI

```
dotnet-counters monitor -n WebMetric --counters Contoso.Web
```

Output similar to the following is displayed:

.NET CLI

```

Press p to pause, r to resume, q to quit.
Status: Running

```

[Contoso.Web]

```

    contoso.product.sold (Count / 1 sec)
        contoso.product.name=Eggs           12
        contoso.product.name=Milk           0

```

# View metrics in Grafana with OpenTelemetry and Prometheus

## Overview

[OpenTelemetry](#) <sup>↗</sup>:

- Is a vendor-neutral open-source project supported by the [Cloud Native Computing Foundation](#) <sup>↗</sup>.
- Standardizes generating and collecting telemetry for cloud-native software.
- Works with .NET using the .NET metric APIs.
- Is endorsed by [Azure Monitor](#) and many APM vendors.

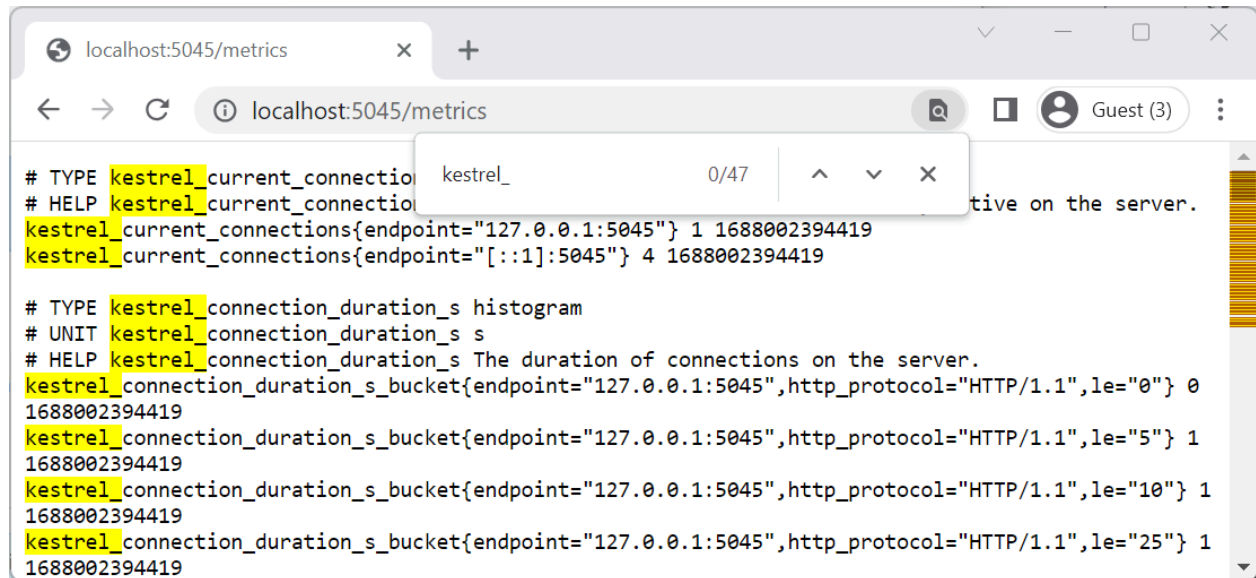
This tutorial shows one of the integrations available for OpenTelemetry metrics using the OSS [Prometheus](#) <sup>↗</sup> and [Grafana](#) <sup>↗</sup> projects. The metrics data flow:

1. The ASP.NET Core metric APIs record measurements from the example app.
2. The OpenTelemetry .NET library running in the app aggregates the measurements.
3. The Prometheus exporter library makes the aggregated data available via an HTTP metrics endpoint. 'Exporter' is what OpenTelemetry calls the libraries that transmit telemetry to vendor-specific backends.
4. A Prometheus server:
  - Polls the metrics endpoint
  - Reads the data
  - Stores the data in a database for long-term persistence. Prometheus refers to reading and storing data as *scraping* an endpoint.
  - Can run on a different machine
5. The Grafana server:
  - Queries the data stored in Prometheus and displays it on a web-based monitoring dashboard.
  - Can run on a different machine.

## View metrics from sample app

Navigate to the sample app. The browser displays `Hello OpenTelemetry! ticks: <3digits>` where `3digits` are the last 3 digits of the current [DateTime.Ticks](#).

Append `/metrics` to the URL to view the metrics endpoint. The browser displays the metrics being collected:



```
# TYPE kestrel_current_connection_info info
# HELP kestrel_current_connection_info Kestrel current connections on the server.
kestrel_current_connections{endpoint="127.0.0.1:5045"} 1 1688002394419
kestrel_current_connections{endpoint="[:,1]:5045"} 4 1688002394419

# TYPE kestrel_connection_duration_s histogram
# UNIT kestrel_connection_duration_s s
# HELP kestrel_connection_duration_s The duration of connections on the server.
kestrel_connection_duration_s_bucket{endpoint="127.0.0.1:5045",http_protocol="HTTP/1.1",le="0"} 0 1688002394419
kestrel_connection_duration_s_bucket{endpoint="127.0.0.1:5045",http_protocol="HTTP/1.1",le="5"} 1 1688002394419
kestrel_connection_duration_s_bucket{endpoint="127.0.0.1:5045",http_protocol="HTTP/1.1",le="10"} 1 1688002394419
kestrel_connection_duration_s_bucket{endpoint="127.0.0.1:5045",http_protocol="HTTP/1.1",le="25"} 1 1688002394419
```

## Set up and configure Prometheus

Follow the [Prometheus first steps](#) to set up a Prometheus server and confirm it's working.

Modify the `prometheus.yml` configuration file so that Prometheus scrapes the metrics endpoint that the example app is exposing. Add the following highlighted text in the `scrape_configs` section:

YAML

```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds.
  Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is
  every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
      - targets:
        # - alertmanager:9093

# Load rules once and periodically evaluate them according to the global
'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"
```

```

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries
  scraped from this config.
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ["localhost:9090"]

- job_name: 'MyASPNETApp'
  scrape_interval: 5s # Poll every 5 seconds for a more responsive demo.
  static_configs:
    - targets: ["localhost:5045"] ## Enter the HTTP port number of the
    demo app.

```

In the preceding highlighted YAML, replace `5045` with the port number that the example app is running on.

## Start Prometheus

1. Reload the configuration or restart the Prometheus server.
2. Confirm that OpenTelemetryTest is in the UP state in the **Status > Targets** page of the Prometheus web portal.

Browser tabs: Prometheus Time Series Collector, localhost:5045/metrics

Address bar: localhost:9090/targets?search=

Prometheus

## Targets

All scrape pools ▾ All Unhealthy Collapse All

Unknown Unhealthy Healthy

**MyASPNETApp (1/1 up)** [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
<a href="http://localhost:5045/metrics">http://localhost:5045/metrics</a>	UP	<code>instance="localhost:5045"</code> <code>job="MyASPNETApp"</code>	543.000 ms ago	2.708ms	

**prometheus (1/1 up)** [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
<a href="http://localhost:9090/metrics">http://localhost:9090/metrics</a>	UP	<code>instance="localhost:9090"</code> <code>job="prometheus"</code>	747.000 ms ago	13.426ms	

Select the **Open metric explorer** icon to see available metrics:

Browser tabs: Prometheus Time Series Collector


Address bar: localhost:9090/graph?g0.expr=current\_requests&g0.tab=0&g0...

Prometheus

☐ Use local time ☐ Enable query history ☒ Enable autocomplete

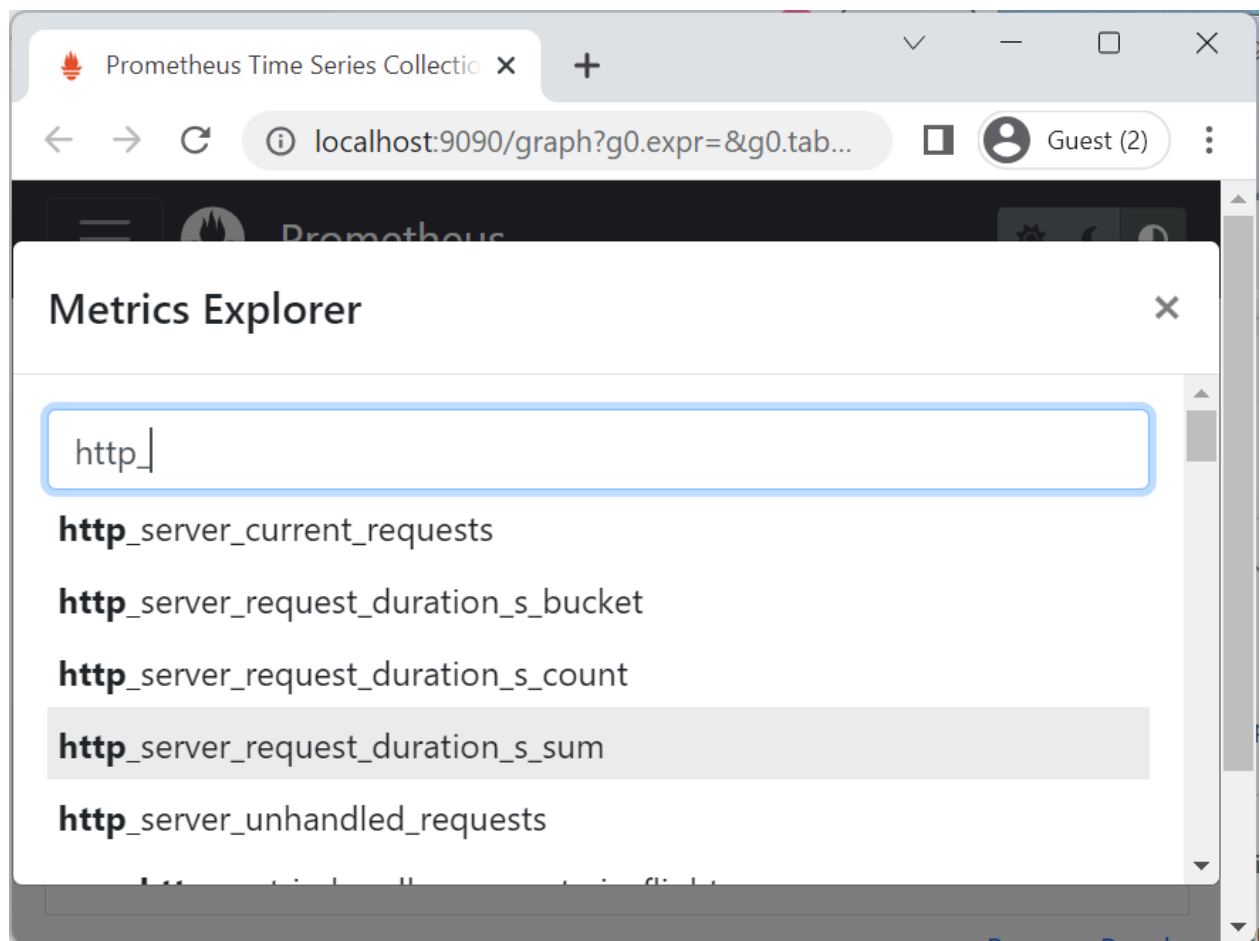
☒ Enable highlighting ☒ Enable linter

Expression (press Shift+Enter for newlines)

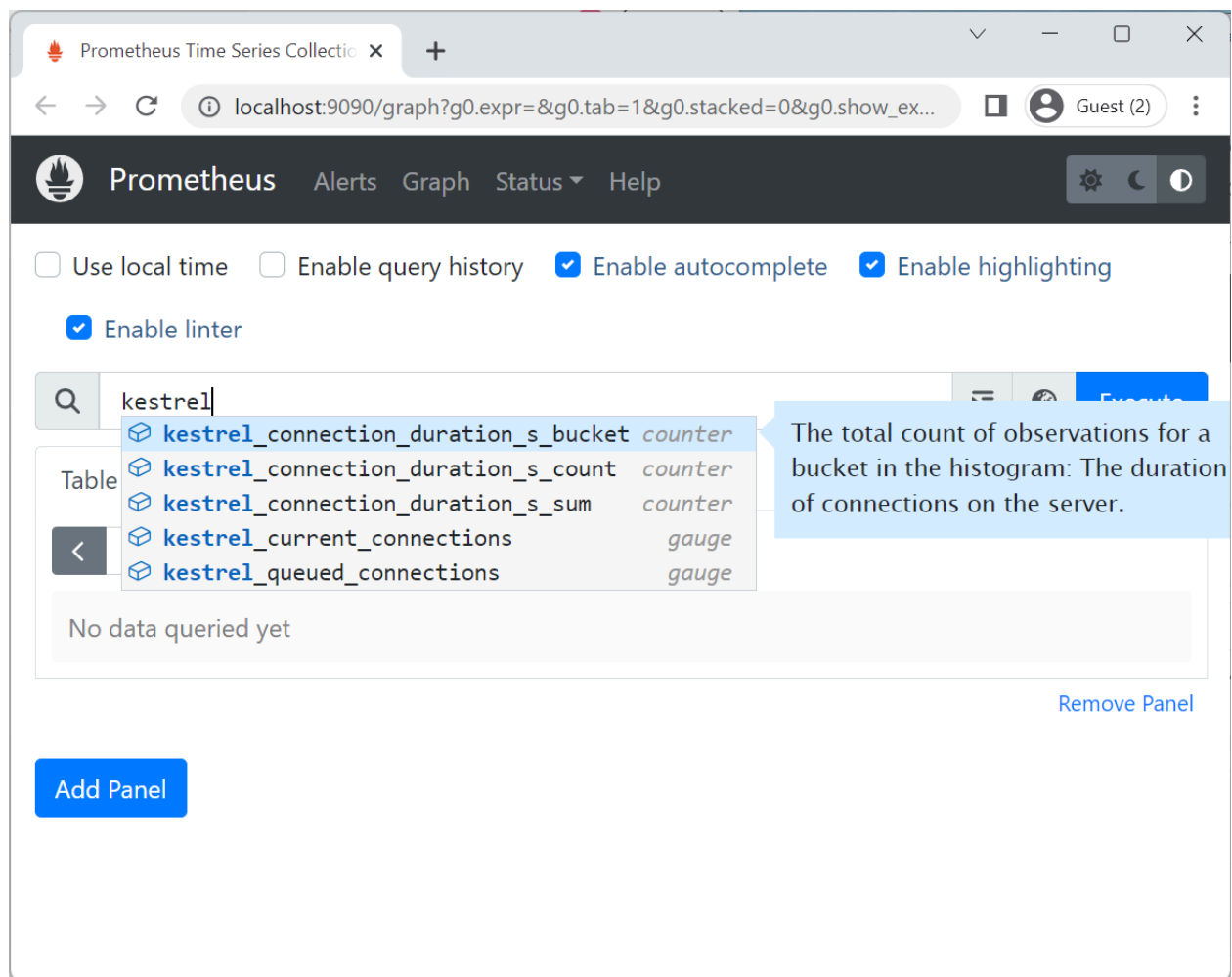
 Execute

Enter counter category such as `http_` in the **Expression** input box to see the available metrics:



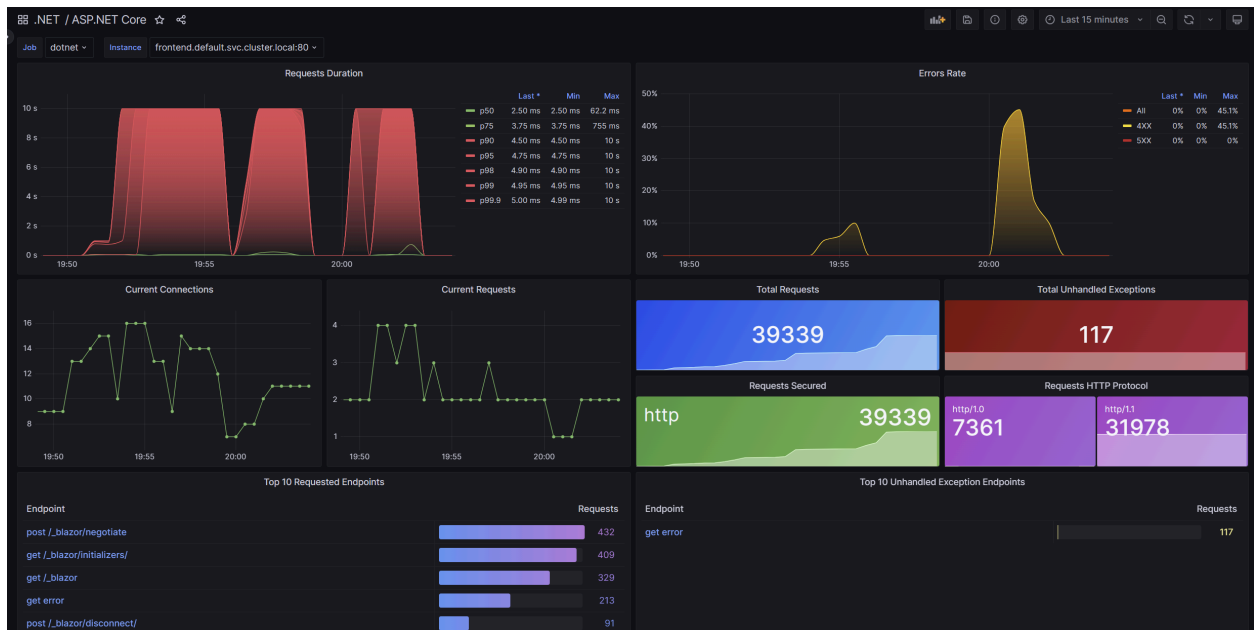


Alternatively, enter counter category such as `kestrel1` in the **Expression** input box to see the available metrics:



## Show metrics on a Grafana dashboard

- Follow the [installation instructions](#) to install Grafana and connect it to a Prometheus data source.
- Follow [Creating a Prometheus graph](#). Alternatively, pre-built dashboards for .NET metrics are available to download at [.NET team dashboards @ grafana.com](#). Downloaded dashboard JSON can be [imported into Grafana](#).



## Test metrics in ASP.NET Core apps

It's possible to test metrics in ASP.NET Core apps. One way to do that is collect and assert metrics values in [ASP.NET Core integration tests](#) using `MetricCollector<T>`.

C#

```
public class BasicTests : IClassFixture<WebApplicationFactory<Program>>
{
    private readonly WebApplicationFactory<Program> _factory;
    public BasicTests(WebApplicationFactory<Program> factory) => _factory =
factory;

    [Fact]
    public async Task Get_RequestCounterIncreased()
    {
        // Arrange
        var client = _factory.CreateClient();
        var meterFactory =
_factory.Services.GetRequiredService<IMeterFactory>();
        var collector = new MetricCollector<double>(meterFactory,
            "Microsoft.AspNetCore.Hosting", "http.server.request.duration");

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

        // Assert
        Assert.Contains("Hello OpenTelemetry!", await
response.Content.ReadAsStringAsync());

        await collector.WaitForMeasurementsAsync(minCount:
1).WaitAsync(TimeSpan.FromSeconds(5));
        Assert.Collection(collector.GetMeasurementSnapshot(),
            measurement =>
```

```

        {
            Assert.Equal("http", measurement.Tags["url.scheme"]);
            Assert.Equal("GET",
measurement.Tags["http.request.method"]);
            Assert.Equal("/", measurement.Tags["http.route"]);
        });
    }
}

```

The proceeding test:

- Bootstraps a web app in memory with [WebApplicationFactory<TEnterPoint>](#). `Program` in the factory's generic argument specifies the web app.
- Collects metrics values with [MetricCollector<T>](#)
  - Requires a package reference to `Microsoft.Extensions.Diagnostics.Testing`
  - The `MetricCollector<T>` is created using the web app's [IMeterFactory](#). This allows the collector to only report metrics values recorded by test.
  - Includes the meter name, `Microsoft.AspNetCore.Hosting`, and counter name, `http.server.request.duration` to collect.
- Makes an HTTP request to the web app.
- Asserts the test using results from the metrics collector.

## ASP.NET Core meters and counters

See [ASP.NET Core metrics](#) for a list of ASP.NET Core meters and counters.

# ASP.NET Core metrics

Article • 02/05/2024

This article describes the metrics built-in for ASP.NET Core produced using the [System.Diagnostics.Metrics](#) API. For a listing of metrics based on the older [EventCounters](#) API, see [here](#).

## 💡 Tip

For more information about how to collect, report, enrich, and test ASP.NET Core metrics, see [Using ASP.NET Core metrics](#).


## Microsoft.AspNetCore.Hosting


The `Microsoft.AspNetCore.Hosting` metrics report high-level information about HTTP requests received by ASP.NET Core:


- [http.server.request.duration](#)
- [http.server.active\\_requests](#)

**Metric:** `http.server.request.duration`

 Expand table

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">http.server.request.duration</a> 	Histogram	s	Measures the duration of inbound HTTP requests.

 Expand table

Attribute	Type	Description	Examples	Presence
<code>http.route</code>	string	The matched route.	<code>{controller}/{action}/{id?}</code>	If it's available.
<code>error.type</code>	string	Describes a class of error the operation ended with.	<code>timeout</code> ; <code>name_resolution_error</code> ; <code>500</code>	If request has ended with an error.
<code>http.request.method</code>	string	HTTP request method.	<code>GET</code> ; <code>POST</code> ; <code>HEAD</code>	Always
<code>http.response.status_code</code>	int	<a href="#">HTTP response status code</a>  .	<code>200</code>	If one was sent.
<code>network.protocol.version</code>	string	Version of the protocol specified in <code>network.protocol.name</code> .	<code>3.1.1</code>	Always

Attribute	Type	Description	Examples	Presence
<code>url.scheme</code>	string	The <a href="#">URI scheme</a> component identifying the used protocol.	<code>http</code> ; <code>https</code>	Always
<code>aspnetcore.request.is_unhandled</code>	Boolean	True when the request wasn't handled by the application pipeline.	<code>true</code>	If the request was unhandled.

The time used to handle an inbound HTTP request as measured at the hosting layer of ASP.NET Core. The time measurement starts once the underlying web host has:

- Sufficiently parsed the HTTP request headers on the inbound network stream to identify the new request.
- Initialized the context data structures such as the [HttpContext](#).

The time ends when:

- The ASP.NET Core handler pipeline is finished executing.
- All response data has been sent.
- The context data structures for the request are being disposed.

When using OpenTelemetry, the default buckets for this metric are set to [ 0.005, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10 ].

Available starting in: .NET 8.0.

**Metric:** `http.server.active_requests`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">http.server.active_requests</a>	UpDownCounter	<code>{request}</code>	Measures the number of concurrent HTTP requests that are currently in-flight.

[Expand table](#)

Attribute	Type	Description	Examples	Presence
<code>http.request.method</code>	string	HTTP request method. [1]	<code>GET</code> ; <code>POST</code> ; <code>HEAD</code>	Always
<code>url.scheme</code>	string	The <a href="#">URI scheme</a> component identifying the used protocol.	<code>http</code> ; <code>https</code>	Always

Available starting in: .NET 8.0.

## Microsoft.AspNetCore.Routing

The `Microsoft.AspNetCore.Routing` metrics report information about [routing HTTP requests](#) to ASP.NET Core endpoints:

- [aspnetcore.routing.match\\_attempts](#)

Metric: `aspnetcore.routing.match_attempts`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">aspnetcore.routing.match_attempts</a> <a href="#">↗</a>	Counter	<code>{match_attempt}</code>	Number of requests that were attempted to be matched to an endpoint.

[Expand table](#)

Attribute	Type	Description	Examples	Presence
<code>aspnetcore.routing.match_status</code>	string	Match result	<code>success</code> ; <code>failure</code>	Always
<code>aspnetcore.routing.is_fallback_route</code>	boolean	A value that indicates whether the matched route is a fallback route.	<code>True</code>	If a route was successfully matched.
<code>http.route</code>	string	The matched route	<code>{controller}/{action}/{id?}</code>	If a route was successfully matched.

Available starting in: .NET 8.0.

## Microsoft.AspNetCore.Diagnostics


The `Microsoft.AspNetCore.Diagnostics` metrics report diagnostics information from [ASP.NET Core error handling middleware](#):

- [aspnetcore.diagnostics.exceptions](#)

Metric: `aspnetcore.diagnostics.exceptions`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">aspnetcore.diagnostics.exceptions</a> ↗	Counter	{exception}	Number of exceptions caught by exception handling middleware.

 Expand table

Attribute	Type	Description	Examples	Presence
<code>aspnetcore.diagnostics.exception.result</code>	string	ASP.NET Core exception middleware handling result	<code>handled</code> ; <code>unhandled</code>	Always
<code>aspnetcore.diagnostics.handler.type</code>	string	Full type name of the <a href="#">IExceptionHandler</a> implementation that handled the exception.	<code>Contoso.MyHandler</code>	If the exception was handled by this handler.
<code>exception.type</code>	string	The full name of exception type.	<code>System.OperationCanceledException</code> ; <code>Contoso.MyException</code>	Always


Available starting in: .NET 8.0.

## Microsoft.AspNetCore.RateLimiting


The `Microsoft.AspNetCore.RateLimiting` metrics report rate limiting information from [ASP.NET Core rate-limiting middleware](#):

- [aspnetcore.rate\\_limiting.active\\_request\\_leases](#)
- [aspnetcore.rate\\_limiting.request\\_lease.duration](#)
- [aspnetcore.rate\\_limiting.queued\\_requests](#)
- [aspnetcore.rate\\_limiting.request.time\\_in\\_queue](#)
- [aspnetcore.rate\\_limiting.requests](#)

Metric: `aspnetcore.rate_limiting.active_request_leases`

 Expand table

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">aspnetcore.rate_limiting.active_request_leases</a> ↗	UpDownCounter	{request}	Number of requests that are currently active on the server that hold a rate limiting lease.


 Expand table





Attribute	Type	Description	Examples	Presence
<code>aspnetcore.rate_limiting.policy</code>	string	Rate limiting policy name.	<code>fixed</code> ; <code>sliding</code> ; <code>token</code>	If the matched endpoint for the request had a rate-limiting policy.

Available starting in: .NET 8.0.

**Metric:** `aspnetcore.rate_limiting.request_lease.duration`

 Expand table


Name	Instrument Type	Unit (UCUM)	Description
<a href="#">aspnetcore.rate_limiting.request_lease.duration</a> 	Histogram	s	The duration of the rate limiting lease held by requests on the server.


 Expand table


Attribute	Type	Description	Examples	Presence
<code>aspnetcore.rate_limiting.policy</code>	string	Rate limiting policy name.	<code>fixed</code> ; <code>sliding</code> ; <code>token</code>	If the matched endpoint for the request had a rate-limiting policy.

Available starting in: .NET 8.0.

**Metric:** `aspnetcore.rate_limiting.queued_requests`

 Expand table

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">aspnetcore.rate_limiting.queued_requests</a> 	UpDownCounter	{request}	Number of requests that are currently queued waiting to acquire a rate limiting lease.


 Expand table

Attribute	Type	Description	Examples	Presence
<code>aspnetcore.rate_limiting.policy</code>	string	Rate limiting policy name.	<code>fixed</code> ; <code>sliding</code> ; <code>token</code>	If the matched endpoint for the request had a rate-limiting policy.

Available starting in: .NET 8.0.

## Metric: `aspnetcore.rate_limiting.request.time_in_queue`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">aspnetcore.rate_limiting.request.time_in_queue</a> 	Histogram	s	The time a request spent in a queue waiting to acquire a rate limiting lease.


[Expand table](#)

Attribute	Type	Description	Examples	Presence
<code>aspnetcore.rate_limiting.policy</code>	string	Rate limiting policy name.	<code>fixed</code> ; <code>sliding</code> ; <code>token</code>	If the matched endpoint for the request had a rate-limiting policy.
<code>aspnetcore.rate_limiting.result</code>	string	The rate limiting result shows whether lease was acquired or contains a rejection reason.	<code>acquired</code> ; <code>request_canceled</code>	Always

Available starting in: .NET 8.0.

## Metric: `aspnetcore.rate_limiting.requests`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">aspnetcore.rate_limiting.requests</a> 	Counter	{request}	Number of requests that tried to acquire a rate limiting lease.

[Expand table](#)

Attribute	Type	Description	Examples	Presence
<code>aspnetcore.rate_limiting.policy</code>	string	Rate limiting policy name.	<code>fixed</code> ; <code>sliding</code> ; <code>token</code>	If the matched endpoint for the request had a rate-limiting policy.
<code>aspnetcore.rate_limiting.result</code>	string	The rate limiting result shows whether lease was acquired or contains a rejection reason.	<code>acquired</code> ; <code>request_canceled</code>	Always

Available starting in: .NET 8.0.

# Microsoft.AspNetCore.HeaderParsing

The `Microsoft.AspNetCore.HeaderParsing` metrics report information about [ASP.NET Core header parsing](#) <sup>↗</sup>:

- [aspnetcore.header\\_parsing.parse\\_errors](#)
- [aspnetcore.header\\_parsing.cache\\_accesses](#)

Metric: `aspnetcore.header_parsing.parse_errors`

 Expand table

Name	Instrument Type	Unit (UCUM)	Description
<code>aspnetcore.header_parsing.parse_errors</code>	Counter	<code>{parse_error}</code>	Number of errors that occurred when parsing HTTP request headers.

 Expand table

Attribute	Type	Description	Examples	Presence
<code>aspnetcore.header_parsing.header.name</code>	string	The header name.	<code>Content-Type</code>	Always
<code>error.type</code>	string	The error message.	<code>Unable to parse media type value.</code>	Always

Available starting in: .NET 8.0.

Metric: `aspnetcore.header_parsing.cache_accesses`

The metric is emitted only for HTTP request header parsers that support caching.

 Expand table

Name	Instrument Type	Unit (UCUM)	Description
<code>aspnetcore.header_parsing.cache_accesses</code>	Counter	<code>{cache_access}</code>	Number of times a cache storing parsed header values was accessed.

 Expand table

Attribute	Type	Description	Examples	Presence
<code>aspnetcore.header_parsing.header.name</code>	string	The header name.	<code>Content-Type</code>	Always

Attribute	Type	Description	Examples	Presence
<code>aspnetcore.header_parsing.cache_access.type</code>	string	A value indicating whether the header's value was found in the cache or not.	<code>Hit</code> ; <code>Miss</code>	Always


Available starting in: .NET 8.0.


## Microsoft.AspNetCore.Server.Kestrel

The `Microsoft.AspNetCore.Server.Kestrel` metrics report HTTP connection information from [ASP.NET Core Kestrel web server](#):




- [kestrel.active\\_connections](#)
- [kestrel.connection.duration](#)
- [kestrel.rejected\\_connections](#)
- [kestrel.queued\\_connections](#)
- [kestrel.queued\\_requests](#)
- [kestrel.upgraded\\_connections](#)
- [kestrel.tls\\_handshake.duration](#)
- [kestrel.active\\_tls\\_handshakes](#)

Metric: `kestrel.active_connections`

 Expand table

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">kestrel.active_connections</a> 	UpDownCounter	<code>{connection}</code>	Number of connections that are currently active on the server.

 Expand table

Attribute	Type	Description	Examples	Presence
<code>network.transport</code>	string	<a href="#">OSI transport layer</a>  or <a href="#">inter-process communication method</a>  .	<code>tcp</code> ; <code>unix</code>	Always
<code>network.type</code>	string	<a href="#">OSI network layer</a>  or non-OSI equivalent.	<code>ipv4</code> ; <code>ipv6</code>	If the transport is <code>tcp</code> or <code>udp</code> .
<code>server.address</code>	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	<code>example.com</code>	Always
<code>server.port</code>	int	Server port number	<code>80</code> ; <code>8080</code> ; <code>443</code>	If the transport is <code>tcp</code> or <code>udp</code> .

Available starting in: .NET 8.0.

## Metric: `kestrel.connection.duration`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">kestrel.connection.duration</a> <a href="#">↗</a>	Histogram	s	The duration of connections on the server.

[Expand table](#)

Attribute	Type	Description	Examples	Presence
<code>error.type</code>	string	Describes a type of error the connection ended with, or the unhandled exception type thrown during the connection pipeline. Known connection errors can be found at <a href="#">Semantic Conventions for Kestrel web server metrics</a> <a href="#">↗</a> .	<code>connection_reset</code> ; <code>invalid_request_headers</code> ; <code>System.OperationCanceledException</code>	If the connection ended with a known error or an exception was thrown.
<code>network.protocol.name</code>	string	<a href="#">OSI application layer</a> <a href="#">↗</a> or non-OSI equivalent.	<code>http</code> ; <code>web_sockets</code>	Always
<code>network.protocol.version</code>	string	Version of the protocol specified in <code>network.protocol.name</code> .	<code>1.1</code> ; <code>2</code>	Always
<code>network.transport</code>	string	<a href="#">OSI transport layer</a> <a href="#">↗</a> or <a href="#">inter-process communication method</a> <a href="#">↗</a> .	<code>tcp</code> ; <code>unix</code>	Always
<code>network.type</code>	string	<a href="#">OSI network layer</a> <a href="#">↗</a> or non-OSI equivalent.	<code>ipv4</code> ; <code>ipv6</code>	If the transport is <code>tcp</code> or <code>udp</code> .
<code>server.address</code>	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	<code>example.com</code>	Always
<code>server.port</code>	int	Server port number	<code>80</code> ; <code>8080</code> ; <code>443</code>	If the transport is <code>tcp</code> or <code>udp</code> .
<code>tls.protocol.version</code>	string	TLS protocol version.	<code>1.2</code> ; <code>1.3</code>	If the connection is secured with TLS.

As this metric is tracking the connection duration, and ideally http connections are used for multiple requests, the buckets should be longer than those used for request durations. For example, using [ 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 30, 60, 120, 300] provides an upper bucket of 5 mins.

Starting in .NET 9, when a connection ends with a known error, the `error.type` attribute value is set to the known error type. Known connection errors can be found at [Semantic Conventions for Kestrel web server metrics](#).

Available starting in: .NET 8.

**Metric:** `kestrel.rejected_connections`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">kestrel.rejected_connections</a>	Counter	{connection}	Number of connections rejected by the server.

[Expand table](#)

Attribute	Type	Description	Examples	Presence
<code>network.transport</code>	string	<a href="#">OSI transport layer</a> or <a href="#">inter-process communication method</a> .	<code>tcp</code> ; <code>unix</code>	Always
<code>network.type</code>	string	<a href="#">OSI network layer</a> or non-OSI equivalent.	<code>ipv4</code> ; <code>ipv6</code>	If the transport is <code>tcp</code> or <code>udp</code> .
<code>server.address</code>	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	<code>example.com</code>	Always
<code>server.port</code>	int	Server port number	<code>80</code> ; <code>8080</code> ; <code>443</code>	If the transport is <code>tcp</code> or <code>udp</code> .

Connections are rejected when the currently active count exceeds the value configured with `MaxConcurrentConnections`.

Available starting in: .NET 8.0.

**Metric:** `kestrel.queued_connections`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">kestrel.queued_connections</a>	UpDownCounter	{connection}	Number of connections that are currently queued and are waiting to start.

[Expand table](#)

Attribute	Type	Description	Examples	Presence
<code>network.transport</code>	string	<a href="#">OSI transport layer</a> or <a href="#">inter-process communication method</a> .	<code>tcp</code> ; <code>unix</code>	Always
<code>network.type</code>	string	<a href="#">OSI network layer</a> or non-OSI equivalent.	<code>ipv4</code> ; <code>ipv6</code>	If the transport is <code>tcp</code> or <code>udp</code> .
<code>server.address</code>	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	<code>example.com</code>	Always
<code>server.port</code>	int	Server port number	<code>80</code> ; <code>8080</code> ; <code>443</code>	If the transport is <code>tcp</code> or <code>udp</code> .

Available starting in: .NET 8.0.

**Metric:** `kestrel.queued_requests`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">kestrel.queued_requests</a>	UpDownCounter	{request}	Number of HTTP requests on multiplexed connections (HTTP/2 and HTTP/3) that are currently queued and are waiting to start.

[Expand table](#)

Attribute	Type	Description	Examples	Presence
<code>network.protocol.name</code>	string	<a href="#">OSI application layer</a> or non-OSI equivalent.	<code>http</code> ; <code>web_sockets</code>	Always
<code>network.protocol.version</code>	string	Version of the protocol specified in <code>network.protocol.name</code> .	<code>1.1</code> ; <code>2</code>	Always
<code>network.transport</code>	string	<a href="#">OSI transport layer</a> or <a href="#">inter-process communication method</a> .	<code>tcp</code> ; <code>unix</code>	Always
<code>network.type</code>	string	<a href="#">OSI network layer</a> or non-OSI equivalent.	<code>ipv4</code> ; <code>ipv6</code>	If the transport is <code>tcp</code> or <code>udp</code> .
<code>server.address</code>	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	<code>example.com</code>	Always
<code>server.port</code>	int	Server port number	<code>80</code> ; <code>8080</code> ; <code>443</code>	If the transport is <code>tcp</code> or <code>udp</code> .

Available starting in: .NET 8.0.

**Metric:** `kestrel.uptime.connections`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">kestrel.uptime.connections</a>	UpDownCounter	{connection}	Number of connections that are currently upgraded (WebSockets).

[Expand table](#)

Attribute	Type	Description	Examples	Presence
<code>network.transport</code>	string	<a href="#">OSI transport layer</a> or <a href="#">inter-process communication method</a> .	<code>tcp</code> ; <code>unix</code>	Always
<code>network.type</code>	string	<a href="#">OSI network layer</a> or non-OSI equivalent.	<code>ipv4</code> ; <code>ipv6</code>	If the transport is <code>tcp</code> or <code>udp</code> .
<code>server.address</code>	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	<code>example.com</code>	Always
<code>server.port</code>	int	Server port number	<code>80</code> ; <code>8080</code> ; <code>443</code>	If the transport is <code>tcp</code> or <code>udp</code> .

The counter only tracks HTTP/1.1 connections.

Available starting in: .NET 8.0.

**Metric:** `kestrel.tls_handshake.duration`

[Expand table](#)

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">kestrel.tls_handshake.duration</a>	Histogram	s	The duration of TLS handshakes on the server.

[Expand table](#)

Attribute	Type	Description	Examples	Presence
<code>error.type</code>	string	The full name of exception type.	<code>System.OperationCanceledException</code> ; <code>Contoso.MyException</code>	If an exception was thrown.
<code>network.transport</code>	string	<a href="#">OSI transport layer</a> or <a href="#">inter-process</a>	<code>tcp</code> ; <code>unix</code>	Always



Attribute	Type	Description	Examples	Presence
		<a href="#">communication method</a> <a href="#">↗</a> .		
<code>network.type</code>	string	<a href="#">OSI network layer</a> <a href="#">↗</a> or non-OSI equivalent.	<code>ipv4</code> ; <code>ipv6</code>	If the transport is <code>tcp</code> or <code>udp</code> .
<code>server.address</code>	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	<code>example.com</code>	Always
<code>server.port</code>	int	Server port number	<code>80</code> ; <code>8080</code> ; <code>443</code>	If the transport is <code>tcp</code> or <code>udp</code> .
<code>tls.protocol.version</code>	string	TLS protocol version.	<code>1.2</code> ; <code>1.3</code>	If the connection is secured with TLS.

When using OpenTelemetry, the default buckets for this metric are set to [ 0.005, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10 ].

Available starting in: .NET 8.0.

**Metric:** `kestrel.active_tls_handshakes`

[↗](#) Expand table

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">kestrel.active_tls_handshakes</a> <a href="#">↗</a>	UpDownCounter	<code>{handshake}</code>	Number of TLS handshakes that are currently in progress on the server.

[↗](#) Expand table

Attribute	Type	Description	Examples	Presence
<code>network.transport</code>	string	<a href="#">OSI transport layer</a> <a href="#">↗</a> or <a href="#">inter-process communication method</a> <a href="#">↗</a> .	<code>tcp</code> ; <code>unix</code>	Always
<code>network.type</code>	string	<a href="#">OSI network layer</a> <a href="#">↗</a> or non-OSI equivalent.	<code>ipv4</code> ; <code>ipv6</code>	If the transport is <code>tcp</code> or <code>udp</code> .
<code>server.address</code>	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	<code>example.com</code>	Always

Attribute	Type	Description	Examples	Presence
<code>server.port</code>	int	Server port number	<code>80</code> ; <code>8080</code> ; <code>443</code>	If the transport is <code>tcp</code> or <code>udp</code> .


Available starting in: .NET 8.0.


# Microsoft.AspNetCore.Http.Connections


The `Microsoft.AspNetCore.Http.Connections` metrics report connection information from [ASP.NET Core SignalR](#):


- [signalr.server.connection.duration](#)
- [signalr.server.active\\_connections](#)

Metric: `signalr.server.connection.duration`


 Expand table

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">signalr.server.connection.duration</a> 	Histogram	<code>s</code>	The duration of connections on the server.

 Expand table


Attribute	Type	Description	Examples	Presence
<code>signalr.connection.status</code>	string	SignalR HTTP connection closure status.	<code>app_shutdown</code> ; <code>timeout</code>	Always
<code>signalr.transport</code>	string	<a href="#">SignalR transport type</a> 	<code>web_sockets</code> ; <code>long_polling</code>	Always

Available starting in: .NET 8.0.

 Expand table

Value	Description
<code>normal_closure</code>	The connection was closed normally.
<code>timeout</code>	The connection was closed due to a timeout.
<code>app_shutdown</code>	The connection was closed because the app is shutting down.

`signalr.transport` is one of the following:


 Expand table

Value	Protocol
<code>server_sent_events</code>	<a href="#">server-sent events</a> ↗
<code>long_polling</code>	<a href="#">Long Polling</a>
<code>web_sockets</code>	<a href="#">WebSocket</a> ↗


As this metric is tracking the connection duration, and ideally SignalR connections are durable, the buckets should be longer than those used for request durations. For example, using [0, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 30, 60, 120, 300] provides an upper bucket of 5 mins.

Available starting in: .NET 8.0.

**Metric:** `signalr.server.active_connections`

 Expand table

Name	Instrument Type	Unit (UCUM)	Description
<a href="#">signalr.server.active_connections</a> ↗	UpDownCounter	<code>{connection}</code>	Number of connections that are currently active on the server.

 Expand table

Attribute	Type	Description	Examples	Presence
<code>signalr.connection.status</code>	string	SignalR HTTP connection closure status.	<code>app_shutdown</code> ; <code>timeout</code>	Always
<code>signalr.transport</code>	string	<a href="#">SignalR transport type</a> ↗	<code>web_sockets</code> ; <code>long_polling</code>	Always

Available starting in: .NET 8.0.

# Use HttpContext in ASP.NET Core

Article • 10/30/2024

## 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](#).

`HttpContext` encapsulates all information about an individual HTTP request and response. An `HttpContext` instance is initialized when an HTTP request is received. The `HttpContext` instance is accessible by middleware and app frameworks such as Web API controllers, Razor Pages, SignalR, gRPC, and more.

For more information about accessing the `HttpContext`, see [Access HttpContext in ASP.NET Core](#).

## HttpRequest

`HttpContext.Request` provides access to `HttpRequest`. `HttpRequest` has information about the incoming HTTP request, and it's initialized when an HTTP request is received by the server. `HttpRequest` isn't read-only, and middleware can change request values in the middleware pipeline.

Commonly used members on `HttpRequest` include:

[Expand table](#)

Property	Description	Example
<code>HttpRequest.Path</code>	The request path.	<code>/en/article/getstarted</code>
<code>HttpRequest.Method</code>	The request method.	<code>GET</code>
<code>HttpRequest.Headers</code>	A collection of request headers.	<code>user-agent=Edge</code> <code>x-custom-</code> <code>header=MyValue</code>
<code>HttpRequest.RouteValues</code>	A collection of route values. The collection is set when the request is	<code>language=en</code> <code>article=getstarted</code>

Property	Description	Example
	matched to a route.	
<a href="#">HttpRequest.Query</a>	A collection of query values parsed from <a href="#">QueryString</a> .	<code>filter=hello</code> <code>page=1</code>
<a href="#">HttpRequest.ReadFormAsync()</a>	A method that reads the request body as a form and returns a form values collection. For information about why <code>ReadFormAsync</code> should be used to access form data, see <a href="#">Prefer ReadFormAsync over Request.Form</a> .	<code>email=user@contoso.com</code>
<a href="#">HttpRequest.Body</a>	A <a href="#">Stream</a> for reading the request body.	UTF-8 JSON payload

## Get request headers

[HttpRequest.Headers](#) provides access to the request headers sent with the HTTP request. There are two ways to access headers using this collection:

- Provide the header name to the indexer on the header collection. The header name isn't case-sensitive. The indexer can access any header value.
- The header collection also has properties for getting and setting commonly used HTTP headers. The properties provide a fast, IntelliSense driven way to access headers.

C#

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

app.MapGet("/", (HttpRequest request) =>
{
    var userAgent = request.Headers.UserAgent;
    var customHeader = request.Headers["x-custom-header"];

    return Results.Ok(new { userAgent = userAgent, customHeader =
customHeader });
});

app.Run();
```

For information on efficiently handling headers that appear more than once, see [A brief look at StringValues](#).

# Read request body

An HTTP request can include a request body. The request body is data associated with the request, such as the content of an HTML form, UTF-8 JSON payload, or a file.

`HttpRequest.Body` allows the request body to be read with [Stream](#):

C#

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

app.MapPost("/uploadstream", async (IConfiguration config, HttpContext
context) =>
{
    var filePath = Path.Combine(config["StoredFilePath"],
Path.GetRandomFileName());

    await using var writeStream = File.Create(filePath);
    await context.Request.Body.CopyToAsync(writeStream);
});

app.Run();
```

`HttpRequest.Body` can be read directly or used with other APIs that accept stream.

## ⚠ Note

[Minimal APIs](#) supports binding [HttpRequest.Body](#) directly to a [Stream](#) parameter.

## Enable request body buffering

The request body can only be read once, from beginning to end. Forward-only reading of the request body avoids the overhead of buffering the entire request body and reduces memory usage. However, in some scenarios, there's a need to read the request body multiple times. For example, middleware might need to read the request body and then rewind it so it's available for the endpoint.

The [EnableBuffering](#) extension method enables buffering of the HTTP request body and is the recommended way to enable multiple reads. Because a request can be any size, `EnableBuffering` supports options for buffering large request bodies to disk, or rejecting them entirely.

The middleware in the following example:

- Enables multiple reads with `EnableBuffering`. It must be called before reading the request body.
- Reads the request body.
- Rewinds the request body to the start so other middleware or the endpoint can read it.

C#

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

app.Use(async (context, next) =>
{
    context.Request.EnableBuffering();
    await ReadRequestBody(context.Request.Body);
    context.Request.Body.Position = 0;

    await next.Invoke();
});

app.Run();
```

## BodyReader

An alternative way to read the request body is to use the `HttpRequest.BodyReader` property. The `BodyReader` property exposes the request body as a `PipeReader`. This API is from [I/O pipelines](#), an advanced, high-performance way to read the request body.

The reader directly accesses the request body and manages memory on the caller's behalf. Unlike `HttpRequest.Body`, the reader doesn't copy request data into a buffer. However, a reader is more complicated to use than a stream and should be used with caution.

For information on how to read content from `BodyReader`, see [I/O pipelines PipeReader](#).

## HttpResponse

`HttpContext.Response` provides access to `HttpResponse`. `HttpResponse` is used to set information on the HTTP response sent back to the client.

Commonly used members on `HttpResponse` include:

Property	Description	Example
<a href="#">HttpResponse.StatusCode</a>	The response code. Must be set before writing to the response body.	200
<a href="#">HttpResponse.ContentType</a>	The response <code>content-type</code> header. Must be set before writing to the response body.	application/json
<a href="#">HttpResponse.Headers</a>	A collection of response headers. Must be set before writing to the response body.	server=Kestrel x-custom- header=MyValue
<a href="#">HttpResponse.Body</a>	A <a href="#">Stream</a> for writing the response body.	Generated web page

## Set response headers

[HttpResponse.Headers](#) provides access to the response headers sent with the HTTP response. There are two ways to access headers using this collection:

- Provide the header name to the indexer on the header collection. The header name isn't case-sensitive. The indexer can access any header value.
- The header collection also has properties for getting and setting commonly used HTTP headers. The properties provide a fast, IntelliSense driven way to access headers.

C#

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

app.MapGet("/", (HttpResponse response) =>
{
    response.Headers.CacheControl = "no-cache";
    response.Headers["x-custom-header"] = "Custom value";

    return Results.File(File.OpenRead("helloworld.txt"));
});

app.Run();
```

An app can't modify headers after the response has started. Once the response starts, the headers are sent to the client. A response is started by flushing the response body or calling [HttpResponse.StartAsync\(CancellationToken\)](#). The [HttpResponse.HasStarted](#) property indicates whether the response has started. An error is thrown when attempting to modify headers after the response has started:



System.InvalidOperationException: Headers are read-only, response has already started.

### ⓘ Note

Unless response buffering is enabled, all write operations (for example, [WriteAsync](#)) flush the response body internally and mark the response as started. Response buffering is disabled by default.

## Write response body

An HTTP response can include a response body. The response body is data associated with the response, such as generated web page content, UTF-8 JSON payload, or a file.

[HttpResponse.Body](#) allows the response body to be written with [Stream](#):

C#

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

app.MapPost("/downloadfile", async (IConfiguration config, HttpContext
context) =>
{
    var filePath = Path.Combine(config["StoredFilePath"],
"helloworld.txt");

    await using var fileStream = File.OpenRead(filePath);
    await fileStream.CopyToAsync(context.Response.Body);
});

app.Run();
```

`HttpResponse.Body` can be written directly or used with other APIs that write to a stream.

## BodyWriter

An alternative way to write the response body is to use the [HttpResponse.BodyWriter](#) property. The `BodyWriter` property exposes the response body as a [PipeWriter](#). This API is from [I/O pipelines](#), and it's an advanced, high-performance way to write the response.

The writer provides direct access to the response body and manages memory on the caller's behalf. Unlike `HttpResponse.Body`, the write doesn't copy request data into a

buffer. However, a writer is more complicated to use than a stream and writer code should be thoroughly tested.

For information on how to write content to `BodyWriter`, see [I/O pipelines PipeWriter](#).

## Set response trailers

HTTP/2 and HTTP/3 support response trailers. Trailers are headers sent with the response after the response body is complete. Because trailers are sent after the response body, trailers can be added to the response at any time.

The following code sets trailers using [AppendTrailer](#):

C#

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

app.MapGet("/", (HttpResponse response) =>
{
    // Write body
    response.WriteAsync("Hello world");

    if (response.SupportsTrailers())
    {
        response.AppendTrailer("trailername", "TrailerValue");
    }
});

app.Run();
```

## RequestAborted

The [HttpContext.RequestAborted](#) cancellation token can be used to notify that the HTTP request has been aborted by the client or server. The cancellation token should be passed to long-running tasks so they can be canceled if the request is aborted. For example, aborting a database query or HTTP request to get data to return in the response.

C#

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

var httpClient = new HttpClient();
app.MapPost("/books/{bookId}", async (int bookId, HttpContext context) =>
```

```
{
    var stream = await httpClient.GetStreamAsync(
        $"http://contoso/books/{bookId}.json", context.RequestAborted);

    // Proxy the response as JSON
    return Results.Stream(stream, "application/json");
});

app.Run();
```

The `RequestAborted` cancellation token doesn't need to be used for request body read operations because reads always throw immediately when the request is aborted. The `RequestAborted` token is also usually unnecessary when writing response bodies, because writes immediately no-op when the request is aborted.

In some cases, passing the `RequestAborted` token to write operations can be a convenient way to force a write loop to exit early with an [OperationCanceledException](#). However, it's typically better to pass the `RequestAborted` token into any asynchronous operations responsible for retrieving the response body content instead.

#### ⓘ Note

[Minimal APIs](#) supports binding [HttpContext.RequestAborted](#) directly to a [CancellationTokens](#) parameter.

## Abort()

The [HttpContext.Abort\(\)](#) method can be used to abort an HTTP request from the server. Aborting the HTTP request immediately triggers the [HttpContext.RequestAborted](#) cancellation token and sends a notification to the client that the server has aborted the request.

The middleware in the following example:

- Adds a custom check for malicious requests.
- Aborts the HTTP request if the request is malicious.

C#

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

app.Use(async (context, next) =>
{
```

```

    if (RequestAppearsMalicious(context.Request))
    {
        // Malicious requests don't even deserve an error response (e.g.
400).
        context.Abort();
        return;
    }

    await next.Invoke();
});

app.Run();

```

## User

The [HttpContext.User](#) property is used to get or set the user, represented by [ClaimsPrincipal](#), for the request. The [ClaimsPrincipal](#) is typically set by [ASP.NET Core authentication](#).

C#

```

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

app.MapGet("/user/current", [Authorize] async (HttpContext context) =>
{
    var user = await GetUserAsync(context.User.Identity.Name);
    return Results.Ok(user);
});

app.Run();

```

### ⓘ Note

[Minimal APIs](#) supports binding [HttpContext.User](#) directly to a [ClaimsPrincipal](#) parameter.

## Features

The [HttpContext.Features](#) property provides access to the collection of feature interfaces for the current request. Since the feature collection is mutable even within the context of a request, middleware can be used to modify the collection and add support for additional features. Some advanced features are only available by accessing the associated interface through the feature collection.

The following example:

- Gets [IHttpRequestBodyDataRateFeature](#) from the features collection.
- Sets [MinDataRate](#) to null. This removes the minimum data rate that the request body must be sent by the client for this HTTP request.

C#

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

app.MapGet("/long-running-stream", async (HttpContext context) =>
{
    var feature = context.Features.Get<IHttpRequestBodyDataRateFeature>
());
    if (feature != null)
    {
        feature.MinDataRate = null;
    }

    // await and read long-running stream from request body.
    await Task.Yield();
});

app.Run();
```

For more information about using request features and `HttpContext`, see [Request Features in ASP.NET Core](#).

## HttpContext isn't thread safe

This article primarily discusses using `HttpContext` in request and response flow from Razor Pages, controllers, middleware, etc. Consider the following when using `HttpContext` outside the request and response flow:

- The `HttpContext` is **NOT** thread safe, accessing it from multiple threads can result in exceptions, data corruption and generally unpredictable results.
- The [IHttpContextAccessor](#) interface should be used with caution. As always, the `HttpContext` must **not** be captured outside of the request flow.  
`IHttpContextAccessor`:
  - Relies on [AsyncLocal<T>](#) which can have a negative performance impact on asynchronous calls.
  - Creates a dependency on "ambient state" which can make testing more difficult.
- [IHttpContextAccessor.HttpContext](#) may be `null` if accessed outside of the request flow.

- To access information from `HttpContext` outside the request flow, copy the information inside the request flow. Be careful to copy the actual data and not just references. For example, rather than copying a reference to an `IHeaderDictionary`, copy the relevant header values or copy the entire dictionary key by key before leaving the request flow.
- Don't capture `IHttpContextAccessor.HttpContext` in a constructor.

The following sample logs GitHub branches when requested from the `/branch` endpoint:

C#

```
using System.Text.Json;
using HttpContextInBackgroundThread;
using Microsoft.Net.Http.Headers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpContextAccessor();
builder.Services.AddHostedService<PeriodicBranchesLoggerService>();

builder.Services.AddHttpClient("GitHub", httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");

    // The GitHub API requires two headers. The Use-Agent header is added
    // dynamically through UserAgentHeaderHandler
    httpClient.DefaultRequestHeaders.Add(
        HeaderNames.Accept, "application/vnd.github.v3+json");
}).AddHttpMessageHandler<UserAgentHeaderHandler>();

builder.Services.AddTransient<UserAgentHeaderHandler>();

var app = builder.Build();

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

```

app.MapGet("/branches", async (IHttpClientFactory httpClientFactory,
                                HttpContext context, Logger<Program> logger) =>
{
    var httpClient = httpClientFactory.CreateClient("GitHub");
    var httpResponseMessage = await httpClient.GetAsync(
        "repos/dotnet/AspNetCore.Docs/branches");

    if (!httpResponseMessage.IsSuccessStatusCode)
        return Results.BadRequest();

    await using var contentStream =
        await httpResponseMessage.Content.ReadAsStreamAsync();

    var response = await JsonSerializer.DeserializeAsync<
        IEnumerable<GitHubBranch>>(contentStream);

    app.Logger.LogInformation($"branches request: " +
        $"{JsonSerializer.Serialize(response)}");

    return Results.Ok(response);
});

app.Run();

```

The GitHub API requires two headers. The `User-Agent` header is added dynamically by the `UserAgentHeaderHandler`:

```

C#

using System.Text.Json;
using HttpContextInBackgroundThread;
using Microsoft.Net.Http.Headers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpContextAccessor();
builder.Services.AddHostedService<PeriodicBranchesLoggerService>();

builder.Services.AddHttpClient("GitHub", httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");

    // The GitHub API requires two headers. The Use-Agent header is added
    // dynamically through UserAgentHeaderHandler
    httpClient.DefaultRequestHeaders.Add(
        HeaderNames.Accept, "application/vnd.github.v3+json");
}).AddHttpMessageHandler<UserAgentHeaderHandler>();

builder.Services.AddTransient<UserAgentHeaderHandler>();

var app = builder.Build();

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

```

```

app.MapGet("/branches", async (IHttpClientFactory httpClientFactory,
                                HttpContext context, Logger<Program> logger) =>
{
    var httpClient = httpClientFactory.CreateClient("GitHub");
    var httpResponseMessage = await httpClient.GetAsync(
        "repos/dotnet/AspNetCore.Docs/branches");

    if (!httpResponseMessage.IsSuccessStatusCode)
        return Results.BadRequest();

    await using var contentStream =
        await httpResponseMessage.Content.ReadAsStreamAsync();

    var response = await JsonSerializer.DeserializeAsync
        <IEnumerable<GitHubBranch>>(contentStream);

    app.Logger.LogInformation($" /branches request: " +
        $"{JsonSerializer.Serialize(response)}");

    return Results.Ok(response);
});

app.Run();

```

The `UserAgentHeaderHandler`:

C#

```

using Microsoft.Net.Http.Headers;

namespace HttpContextInBackgroundThread;

public class UserAgentHeaderHandler : DelegatingHandler
{
    private readonly IHttpContextAccessor _httpContextAccessor;
    private readonly ILogger _logger;

    public UserAgentHeaderHandler(IHttpContextAccessor httpContextAccessor,
                                ILogger<UserAgentHeaderHandler> logger)
    {
        _httpContextAccessor = httpContextAccessor;
        _logger = logger;
    }

    protected override async Task<HttpResponseMessage>
        SendAsync(HttpRequestMessage request,
                  CancellationToken cancellationToken)
    {

```



```

        var contextRequest = _httpContextAccessor.HttpContext?.Request;
        string? userAgentString = contextRequest?.Headers["user-agent"].ToString();

        if (string.IsNullOrEmpty(userAgentString))
        {
            userAgentString = "Unknown";
        }

        request.Headers.Add(HeaderNames.UserAgent, userAgentString);
        _logger.LogInformation($"User-Agent: {userAgentString}");

        return await base.SendAsync(request, cancellationToken);
    }
}

```

In the preceding code, when the `HttpContext` is `null`, the `userAgent` string is set to `"Unknown"`. If possible, `HttpContext` should be explicitly passed to the service. Explicitly passing in `HttpContext` data:

- Makes the service API more useable outside the request flow.
- Is better for performance.
- Makes the code easier to understand and reason about than relying on ambient state.

When the service must access `HttpContext`, it should account for the possibility of `HttpContext` being `null` when not called from a request thread.

The application also includes `PeriodicBranchesLoggerService`, which logs the open GitHub branches of the specified repository every 30 seconds:

```

C#

using System.Text.Json;

namespace HttpContextInBackgroundThread;

public class PeriodicBranchesLoggerService : BackgroundService
{
    private readonly IHttpClientFactory _httpClientFactory;
    private readonly ILogger _logger;
    private readonly PeriodicTimer _timer;

    public PeriodicBranchesLoggerService(IHttpClientFactory
httpClientFactory,

ILogger<PeriodicBranchesLoggerService> logger)
    {
        _httpClientFactory = httpClientFactory;
        _logger = logger;
    }
}

```

```

        _timer = new PeriodicTimer(TimeSpan.FromSeconds(30));
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (await _timer.WaitForNextTickAsync(stoppingToken))
        {
            try
            {
                // Cancel sending the request to sync branches if it takes
too long
                // rather than miss sending the next request scheduled 30
seconds from now.
                // Having a single loop prevents this service from sending
an unbounded
                // number of requests simultaneously.
                using var syncTokenSource =
CancellationTokenSource.CreateLinkedTokenSource(stoppingToken);
                syncTokenSource.CancelAfter(TimeSpan.FromSeconds(30));

                var httpClient = _httpClientFactory.CreateClient("GitHub");
                var httpResponseMessage = await
httpClient.GetAsync("repos/dotnet/AspNetCore.Docs/branches",
stoppingToken);

                if (httpResponseMessage.IsSuccessStatusCode)
                {
                    await using var contentStream =
                        await
httpResponseMessage.Content.ReadAsStreamAsync(stoppingToken);

                    // Sync the response with preferred datastore.
                    var response = await JsonSerializer.DeserializeAsync<
IEnumerable<GitHubBranch>>(contentStream,
cancellationTokens: stoppingToken);

                    _logger.LogInformation(
                        $"Branch sync successful! Response:
{JsonSerializer.Serialize(response)}");
                }
                else
                {
                    _logger.LogError(1, $"Branch sync failed! HTTP status
code: {httpResponseMessage.StatusCode}");
                }
            }
            catch (Exception ex)
            {
                _logger.LogError(1, ex, "Branch sync failed!");
            }
        }
    }
}

```

```

    public override Task StopAsync(CancellationToken stoppingToken)
    {
        // This will cause any active call to WaitForNextTickAsync() to
        return false immediately.
        _timer.Dispose();
        // This will cancel the stoppingToken and await
        ExecuteAsync(stoppingToken).
        return base.StopAsync(stoppingToken);
    }
}

```

`PeriodicBranchesLoggerService` is a [hosted service](#), which runs outside the request and response flow. Logging from the `PeriodicBranchesLoggerService` has a null `HttpContext`. The `PeriodicBranchesLoggerService` was written to not depend on the `HttpContext`.

C#

```

using System.Text.Json;
using HttpContextInBackgroundThread;
using Microsoft.Net.Http.Headers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpContextAccessor();
builder.Services.AddHostedService<PeriodicBranchesLoggerService>();

builder.Services.AddHttpClient("GitHub", httpClient =>
{

```

# Routing in ASP.NET Core

Article • 09/18/2024

By [Ryan Nowak](#), [Kirk Larkin](#), and [Rick Anderson](#)

## 📘 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](#).

Routing is responsible for matching incoming HTTP requests and dispatching those requests to the app's executable endpoints. [Endpoints](#) are the app's units of executable request-handling code. Endpoints are defined in the app and configured when the app starts. The endpoint matching process can extract values from the request's URL and provide those values for request processing. Using endpoint information from the app, routing is also able to generate URLs that map to endpoints.

Apps can configure routing using:

- Controllers
- Razor Pages
- SignalR
- gRPC Services
- Endpoint-enabled [middleware](#) such as [Health Checks](#).
- Delegates and lambdas registered with routing.

This article covers low-level details of ASP.NET Core routing. For information on configuring routing:

- For controllers, see [Routing to controller actions in ASP.NET Core](#).
- For Razor Pages conventions, see [Razor Pages route and app conventions in ASP.NET Core](#).
- For Blazor routing guidance, which adds to or supersedes the guidance in this article, see [ASP.NET Core Blazor routing and navigation](#).

## Routing basics

The following code shows a basic example of routing:

C#

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

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

app.Run();
```

The preceding example includes a single endpoint using the [MapGet](#) method:

- When an HTTP `GET` request is sent to the root URL `/`:
  - The request delegate executes.
  - `Hello World!` is written to the HTTP response.
- If the request method is not `GET` or the root URL is not `/`, no route matches and an HTTP 404 is returned.

Routing uses a pair of middleware, registered by [UseRouting](#) and [UseEndpoints](#):

- `UseRouting` adds route matching to the middleware pipeline. This middleware looks at the set of endpoints defined in the app, and selects the [best match](#) based on the request.
- `UseEndpoints` adds endpoint execution to the middleware pipeline. It runs the delegate associated with the selected endpoint.

Apps typically don't need to call `UseRouting` or `UseEndpoints`. [WebApplicationBuilder](#) configures a middleware pipeline that wraps middleware added in `Program.cs` with `UseRouting` and `UseEndpoints`. However, apps can change the order in which `UseRouting` and `UseEndpoints` run by calling these methods explicitly. For example, the following code makes an explicit call to `UseRouting`:

C#

```
app.Use(async (context, next) =>
{
    // ...
    await next(context);
});

app.UseRouting();

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

In the preceding code:

- The call to `app.Use` registers a custom middleware that runs at the start of the pipeline.
- The call to `UseRouting` configures the route matching middleware to run *after* the custom middleware.
- The endpoint registered with `MapGet` runs at the end of the pipeline.

If the preceding example didn't include a call to `UseRouting`, the custom middleware would run *after* the route matching middleware.

**Note:** Routes added directly to the `WebApplication` execute at the *end* of the pipeline.

## Endpoints

The `MapGet` method is used to define an **endpoint**. An endpoint is something that can be:

- Selected, by matching the URL and HTTP method.
- Executed, by running the delegate.

Endpoints that can be matched and executed by the app are configured in `UseEndpoints`. For example, `MapGet`, `MapPost`, and [similar methods](#) connect request delegates to the routing system. Additional methods can be used to connect ASP.NET Core framework features to the routing system:

- [MapRazorPages](#) for Razor Pages
- [MapControllers](#) for controllers
- [MapHub<THub>](#) for SignalR
- [MapGrpcService<TService>](#) for gRPC

The following example shows routing with a more sophisticated route template:

C#

```
app.MapGet("/hello/{name:alpha}", (string name) => $"Hello {name}!");
```

The string `/hello/{name:alpha}` is a **route template**. A route template is used to configure how the endpoint is matched. In this case, the template matches:

- A URL like `/hello/Docs`
- Any URL path that begins with `/hello/` followed by a sequence of alphabetic characters. `:alpha` applies a route constraint that matches only alphabetic characters. [Route constraints](#) are explained later in this article.

The second segment of the URL path, `{name:alpha}`:

- Is bound to the `name` parameter.
- Is captured and stored in `HttpRequest.RouteValues`.

The following example shows routing with [health checks](#) and authorization:

C#

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

app.MapHealthChecks("/healthz").RequireAuthorization();
app.MapGet("/", () => "Hello World!");
```

The preceding example demonstrates how:

- The authorization middleware can be used with routing.
- Endpoints can be used to configure authorization behavior.

The [MapHealthChecks](#) call adds a health check endpoint. Chaining [RequireAuthorization](#) on to this call attaches an authorization policy to the endpoint.

Calling [UseAuthentication](#) and [UseAuthorization](#) adds the authentication and authorization middleware. These middleware are placed between [UseRouting](#) and [UseEndpoints](#) so that they can:

- See which endpoint was selected by [UseRouting](#).
- Apply an authorization policy before [UseEndpoints](#) dispatches to the endpoint.

## Endpoint metadata

In the preceding example, there are two endpoints, but only the health check endpoint has an authorization policy attached. If the request matches the health check endpoint, `/healthz`, an authorization check is performed. This demonstrates that endpoints can have extra data attached to them. This extra data is called endpoint **metadata**:

- The metadata can be processed by routing-aware middleware.
- The metadata can be of any .NET type.

## Routing concepts

The routing system builds on top of the middleware pipeline by adding the powerful **endpoint** concept. Endpoints represent units of the app's functionality that are distinct

from each other in terms of routing, authorization, and any number of ASP.NET Core's systems.

## ASP.NET Core endpoint definition

An ASP.NET Core endpoint is:

- Executable: Has a [RequestDelegate](#).
- Extensible: Has a [Metadata](#) collection.
- Selectable: Optionally, has [routing information](#).
- Enumerable: The collection of endpoints can be listed by retrieving the [EndpointDataSource](#) from [DI](#).

The following code shows how to retrieve and inspect the endpoint matching the current request:

```
C#

app.Use(async (context, next) =>
{
    var currentEndpoint = context.GetEndpoint();

    if (currentEndpoint is null)
    {
        await next(context);
        return;
    }

    Console.WriteLine($"Endpoint: {currentEndpoint.DisplayName}");

    if (currentEndpoint is RouteEndpoint routeEndpoint)
    {
        Console.WriteLine($" - Route Pattern: {routeEndpoint.RoutePattern}");
    }

    foreach (var endpointMetadata in currentEndpoint.Metadata)
    {
        Console.WriteLine($" - Metadata: {endpointMetadata}");
    }

    await next(context);
});

app.MapGet("/", () => "Inspect Endpoint.");
```

The endpoint, if selected, can be retrieved from the `HttpContext`. Its properties can be inspected. Endpoint objects are immutable and cannot be modified after creation. The



most common type of endpoint is a [RouteEndpoint](#). `RouteEndpoint` includes information that allows it to be selected by the routing system.

In the preceding code, `app.Use` configures an inline [middleware](#).

The following code shows that, depending on where `app.Use` is called in the pipeline, there may not be an endpoint:

C#

```
// Location 1: before routing runs, endpoint is always null here.
app.Use(async (context, next) =>
{
    Console.WriteLine($"1. Endpoint: {context.GetEndpoint()?.DisplayName ??
"(null)"}");
    await next(context);
});

app.UseRouting();

// Location 2: after routing runs, endpoint will be non-null if routing
found a match.
app.Use(async (context, next) =>
{
    Console.WriteLine($"2. Endpoint: {context.GetEndpoint()?.DisplayName ??
"(null)"}");
    await next(context);
});

// Location 3: runs when this endpoint matches
app.MapGet("/", (HttpContext context) =>
{
    Console.WriteLine($"3. Endpoint: {context.GetEndpoint()?.DisplayName ??
"(null)"}");
    return "Hello World!";
}).WithDisplayName("Hello");

app.UseEndpoints(_ => { });

// Location 4: runs after UseEndpoints - will only run if there was no
match.
app.Use(async (context, next) =>
{
    Console.WriteLine($"4. Endpoint: {context.GetEndpoint()?.DisplayName ??
"(null)"}");
    await next(context);
});
```

The preceding sample adds `Console.WriteLine` statements that display whether or not an endpoint has been selected. For clarity, the sample assigns a display name to the provided `/` endpoint.

The preceding sample also includes calls to `UseRouting` and `UseEndpoints` to control exactly when these middleware run within the pipeline.

Running this code with a URL of `/` displays:

txt

```
1. Endpoint: (null)
2. Endpoint: Hello
3. Endpoint: Hello
```

Running this code with any other URL displays:

txt

```
1. Endpoint: (null)
2. Endpoint: (null)
4. Endpoint: (null)
```

This output demonstrates that:

- The endpoint is always null before `UseRouting` is called.
- If a match is found, the endpoint is non-null between `UseRouting` and `UseEndpoints`.
- The `UseEndpoints` middleware is **terminal** when a match is found. [Terminal middleware](#) is defined later in this article.
- The middleware after `UseEndpoints` execute only when no match is found.

The `UseRouting` middleware uses the [SetEndpoint](#) method to attach the endpoint to the current context. It's possible to replace the `UseRouting` middleware with custom logic and still get the benefits of using endpoints. Endpoints are a low-level primitive like middleware, and aren't coupled to the routing implementation. Most apps don't need to replace `UseRouting` with custom logic.

The `UseEndpoints` middleware is designed to be used in tandem with the `UseRouting` middleware. The core logic to execute an endpoint isn't complicated. Use [GetEndpoint](#) to retrieve the endpoint, and then invoke its [RequestDelegate](#) property.

The following code demonstrates how middleware can influence or react to routing:

C#

```
app.UseHttpMethodOverride();
app.UseRouting();
```

```

app.Use(async (context, next) =>
{
    if (context.GetEndpoint()?.Metadata.GetMetadata<RequiresAuditAttribute>
() is not null)
    {
        Console.WriteLine($"ACCESS TO SENSITIVE DATA AT:
{DateTime.UtcNow}");
    }

    await next(context);
});

app.MapGet("/", () => "Audit isn't required.");
app.MapGet("/sensitive", () => "Audit required for sensitive data.")
    .WithMetadata(new RequiresAuditAttribute());

```

C#

```

public class RequiresAuditAttribute : Attribute { }

```

The preceding example demonstrates two important concepts:

- Middleware can run before `UseRouting` to modify the data that routing operates upon.
  - Usually middleware that appears before routing modifies some property of the request, such as `UseRewriter`, `UseHttpMethodOverride`, or `UsePathBase`.
- Middleware can run between `UseRouting` and `UseEndpoints` to process the results of routing before the endpoint is executed.
  - Middleware that runs between `UseRouting` and `UseEndpoints`:
    - Usually inspects metadata to understand the endpoints.
    - Often makes security decisions, as done by `UseAuthorization` and `UseCors`.
  - The combination of middleware and metadata allows configuring policies per-endpoint.

The preceding code shows an example of a custom middleware that supports per-endpoint policies. The middleware writes an *audit log* of access to sensitive data to the console. The middleware can be configured to *audit* an endpoint with the `RequiresAuditAttribute` metadata. This sample demonstrates an *opt-in* pattern where only endpoints that are marked as sensitive are audited. It's possible to define this logic in reverse, auditing everything that isn't marked as safe, for example. The endpoint metadata system is flexible. This logic could be designed in whatever way suits the use case.

The preceding sample code is intended to demonstrate the basic concepts of endpoints. **The sample is not intended for production use.** A more complete version of an *audit*

*log* middleware would:

- Log to a file or database.
- Include details such as the user, IP address, name of the sensitive endpoint, and more.

The audit policy metadata `RequiresAuditAttribute` is defined as an `Attribute` for easier use with class-based frameworks such as controllers and SignalR. When using *route to code*:

- Metadata is attached with a builder API.
- Class-based frameworks include all attributes on the corresponding method and class when creating endpoints.

The best practices for metadata types are to define them either as interfaces or attributes. Interfaces and attributes allow code reuse. The metadata system is flexible and doesn't impose any limitations.

## Compare terminal middleware with routing

The following example demonstrates both terminal middleware and routing:

C#

```
// Approach 1: Terminal Middleware.
app.Use(async (context, next) =>
{
    if (context.Request.Path == "/")
    {
        await context.Response.WriteAsync("Terminal Middleware.");
        return;
    }

    await next(context);
});

app.UseRouting();

// Approach 2: Routing.
app.MapGet("/Routing", () => "Routing.");
```

The style of middleware shown with `Approach 1:` is **terminal middleware**. It's called terminal middleware because it does a matching operation:

- The matching operation in the preceding sample is `Path == "/"` for the middleware and `Path == "/Routing"` for routing.

- When a match is successful, it executes some functionality and returns, rather than invoking the `next` middleware.

It's called terminal middleware because it terminates the search, executes some functionality, and then returns.

The following list compares terminal middleware with routing:

- Both approaches allow terminating the processing pipeline:
  - Middleware terminates the pipeline by returning rather than invoking `next`.
  - Endpoints are always terminal.
- Terminal middleware allows positioning the middleware at an arbitrary place in the pipeline:
  - Endpoints execute at the position of `UseEndpoints`.
- Terminal middleware allows arbitrary code to determine when the middleware matches:
  - Custom route matching code can be verbose and difficult to write correctly.
  - Routing provides straightforward solutions for typical apps. Most apps don't require custom route matching code.
- Endpoints interface with middleware such as `UseAuthorization` and `UseCors`.
  - Using a terminal middleware with `UseAuthorization` or `UseCors` requires manual interfacing with the authorization system.

An [endpoint](#) defines both:

- A delegate to process requests.
- A collection of arbitrary metadata. The metadata is used to implement cross-cutting concerns based on policies and configuration attached to each endpoint.

Terminal middleware can be an effective tool, but can require:

- A significant amount of coding and testing.
- Manual integration with other systems to achieve the desired level of flexibility.

Consider integrating with routing before writing a terminal middleware.

Existing terminal middleware that integrates with `Map` or `MapWhen` can usually be turned into a routing aware endpoint. [MapHealthChecks](#) [↗](#) demonstrates the pattern for router-aware:

- Write an extension method on `IEndpointRouteBuilder`.
- Create a nested middleware pipeline using `CreateApplicationBuilder`.
- Attach the middleware to the new pipeline. In this case, `UseHealthChecks`.
- `Build` the middleware pipeline into a `RequestDelegate`.

- Call `Map` and provide the new middleware pipeline.
- Return the builder object provided by `Map` from the extension method.

The following code shows use of `MapHealthChecks`:

C#

```
app.UseAuthentication();  
app.UseAuthorization();  
  
app.MapHealthChecks("/healthz").RequireAuthorization();
```

The preceding sample shows why returning the builder object is important. Returning the builder object allows the app developer to configure policies such as authorization for the endpoint. In this example, the health checks middleware has no direct integration with the authorization system.

The metadata system was created in response to the problems encountered by extensibility authors using terminal middleware. It's problematic for each middleware to implement its own integration with the authorization system.

## URL matching

- Is the process by which routing matches an incoming request to an [endpoint](#).
- Is based on data in the URL path and headers.
- Can be extended to consider any data in the request.

When a routing middleware executes, it sets an `Endpoint` and route values to a [request feature](#) on the `HttpContext` from the current request:

- Calling `HttpContext.GetEndpoint` gets the endpoint.
- `HttpRequest.RouteValues` gets the collection of route values.

[Middleware](#) that runs after the routing middleware can inspect the endpoint and take action. For example, an authorization middleware can interrogate the endpoint's metadata collection for an authorization policy. After all of the middleware in the request processing pipeline is executed, the selected endpoint's delegate is invoked.

The routing system in endpoint routing is responsible for all dispatching decisions. Because the middleware applies policies based on the selected endpoint, it's important that:

- Any decision that can affect dispatching or the application of security policies is made inside the routing system.

### ⚠ Warning

For backward-compatibility, when a Controller or Razor Pages endpoint delegate is executed, the properties of `RouteContext.RouteData` are set to appropriate values based on the request processing performed thus far.

The `RouteContext` type will be marked obsolete in a future release:

- Migrate `RouteData.Values` to `HttpRequest.RouteValues`.
- Migrate `RouteData.DataTokens` to retrieve `IDataTokensMetadata` from the endpoint metadata.

URL matching operates in a configurable set of phases. In each phase, the output is a set of matches. The set of matches can be narrowed down further by the next phase. The routing implementation does not guarantee a processing order for matching endpoints. **All** possible matches are processed at once. The URL matching phases occur in the following order. ASP.NET Core:

1. Processes the URL path against the set of endpoints and their route templates, collecting **all** of the matches.
2. Takes the preceding list and removes matches that fail with route constraints applied.
3. Takes the preceding list and removes matches that fail the set of `MatcherPolicy` instances.
4. Uses the `EndpointSelector` to make a final decision from the preceding list.

The list of endpoints is prioritized according to:

- The `RouteEndpoint.Order`
- The `route template precedence`

All matching endpoints are processed in each phase until the `EndpointSelector` is reached. The `EndpointSelector` is the final phase. It chooses the highest priority endpoint from the matches as the best match. If there are other matches with the same priority as the best match, an ambiguous match exception is thrown.

The route precedence is computed based on a **more specific** route template being given a higher priority. For example, consider the templates `/hello` and `/ {message}`:

- Both match the URL path `/hello`.
- `/hello` is more specific and therefore higher priority.

In general, route precedence does a good job of choosing the best match for the kinds of URL schemes used in practice. Use [Order](#) only when necessary to avoid an ambiguity.

Due to the kinds of extensibility provided by routing, it isn't possible for the routing system to compute ahead of time the ambiguous routes. Consider an example such as the route templates `/message:alpha` and `/message:int`:

- The `alpha` constraint matches only alphabetic characters.
- The `int` constraint matches only numbers.
- These templates have the same route precedence, but there's no single URL they both match.
- If the routing system reported an ambiguity error at startup, it would block this valid use case.


### **Warning**

The order of operations inside [UseEndpoints](#) doesn't influence the behavior of routing, with one exception. [MapControllerRoute](#) and [MapAreaRoute](#) automatically assign an order value to their endpoints based on the order they are invoked. This simulates long-time behavior of controllers without the routing system providing the same guarantees as older routing implementations.

Endpoint routing in ASP.NET Core:

- Doesn't have the concept of routes.
- Doesn't provide ordering guarantees. All endpoints are processed at once.

## Route template precedence and endpoint selection order

[Route template precedence](#)  is a system that assigns each route template a value based on how specific it is. Route template precedence:

- Avoids the need to adjust the order of endpoints in common cases.
- Attempts to match the common-sense expectations of routing behavior.

For example, consider templates `/Products/List` and `/Products/{id}`. It would be reasonable to assume that `/Products/List` is a better match than `/Products/{id}` for the URL path `/Products/List`. This works because the literal segment `/List` is considered to have better precedence than the parameter segment `/id`.

The details of how precedence works are coupled to how route templates are defined:



- Templates with more segments are considered more specific.
- A segment with literal text is considered more specific than a parameter segment.
- A parameter segment with a constraint is considered more specific than one without.
- A complex segment is considered as specific as a parameter segment with a constraint.
- Catch-all parameters are the least specific. See **catch-all** in the [Route templates](#) section for important information on catch-all routes.

## URL generation concepts

URL generation:

- Is the process by which routing can create a URL path based on a set of route values.
- Allows for a logical separation between endpoints and the URLs that access them.

Endpoint routing includes the [LinkGenerator](#) API. `LinkGenerator` is a singleton service available from [DI](#). The `LinkGenerator` API can be used outside of the context of an executing request. [Mvc.UrlHelper](#) and scenarios that rely on [IUrlHelper](#), such as [Tag Helpers](#), [HTML Helpers](#), and [Action Results](#), use the `LinkGenerator` API internally to provide link generating capabilities.

The link generator is backed by the concept of an **address** and **address schemes**. An address scheme is a way of determining the endpoints that should be considered for link generation. For example, the route name and route values scenarios many users are familiar with from controllers and Razor Pages are implemented as an address scheme.

The link generator can link to controllers and Razor Pages via the following extension methods:

- [GetPathByAction](#)
- [GetUriByAction](#)
- [GetPathByPage](#)
- [GetUriByPage](#)

Overloads of these methods accept arguments that include the `HttpContext`. These methods are functionally equivalent to [Url.Action](#) and [Url.Page](#), but offer additional flexibility and options.

The `GetPath*` methods are most similar to `Url.Action` and `Url.Page`, in that they generate a URI containing an absolute path. The `GetUri*` methods always generate an absolute URI containing a scheme and host. The methods that accept an `HttpContext`

generate a URI in the context of the executing request. The `ambient` route values, URL base path, scheme, and host from the executing request are used unless overridden.

`LinkGenerator` is called with an address. Generating a URI occurs in two steps:

1. An address is bound to a list of endpoints that match the address.
2. Each endpoint's `RoutePattern` is evaluated until a route pattern that matches the supplied values is found. The resulting output is combined with the other URI parts supplied to the link generator and returned.

The methods provided by `LinkGenerator` support standard link generation capabilities for any type of address. The most convenient way to use the link generator is through extension methods that perform operations for a specific address type:

 Expand table

Extension Method	Description
<code>GetPathByAddress</code>	Generates a URI with an absolute path based on the provided values.
<code>GetUriByAddress</code>	Generates an absolute URI based on the provided values.

### Warning

Pay attention to the following implications of calling `LinkGenerator` methods:

- Use `GetUri*` extension methods with caution in an app configuration that doesn't validate the `Host` header of incoming requests. If the `Host` header of incoming requests isn't validated, untrusted request input can be sent back to the client in URIs in a view or page. We recommend that all production apps configure their server to validate the `Host` header against known valid values.
- Use `LinkGenerator` with caution in middleware in combination with `Map` or `MapWhen`. `Map*` changes the base path of the executing request, which affects the output of link generation. All of the `LinkGenerator` APIs allow specifying a base path. Specify an empty base path to undo the `Map*` affect on link generation.

## Middleware example

In the following example, a middleware uses the `LinkGenerator` API to create a link to an action method that lists store products. Using the link generator by injecting it into a

class and calling `GenerateLink` is available to any class in an app:

C#

```
public class ProductsMiddleware
{
    private readonly LinkGenerator _linkGenerator;

    public ProductsMiddleware(RequestDelegate next, LinkGenerator
linkGenerator) =>
        _linkGenerator = linkGenerator;

    public async Task InvokeAsync(HttpContext httpContext)
    {
        httpContext.Response.ContentType = MediaTypeNames.Text.Plain;

        var productsPath = _linkGenerator.GetPathByAction("Products",
"Store");

        await httpContext.Response.WriteAsync(
            $"Go to {productsPath} to see our products.");
    }
}
```

## Route templates

Tokens within `{ }` define route parameters that are bound if the route is matched. More than one route parameter can be defined in a route segment, but route parameters must be separated by a literal value. For example:

```
{controller=Home}{action=Index}
```

isn't a valid route, because there's no literal value between `{controller}` and `{action}`. Route parameters must have a name and may have additional attributes specified.

Literal text other than route parameters (for example, `{id}`) and the path separator `/` must match the text in the URL. Text matching is case-insensitive and based on the decoded representation of the URL's path. To match a literal route parameter delimiter `{` or `}`, escape the delimiter by repeating the character. For example `{{` or `}}`.

Asterisk `*` or double asterisk `**`:

- Can be used as a prefix to a route parameter to bind to the rest of the URL.
- Are called a **catch-all** parameters. For example, `blog/{**slug}`:
  - Matches any URI that starts with `blog/` and has any value following it.
  - The value following `blog/` is assigned to the `slug` route value.

Catch-all parameters can also match the empty string.

The catch-all parameter escapes the appropriate characters when the route is used to generate a URL, including path separator `/` characters. For example, the route `foo/{*path}` with route values `{ path = "my/path" }` generates `foo/my%2Fpath`. Note the escaped forward slash. To round-trip path separator characters, use the `**` route parameter prefix. The route `foo/{**path}` with `{ path = "my/path" }` generates `foo/my/path`.

URL patterns that attempt to capture a file name with an optional file extension have additional considerations. For example, consider the template `files/{filename}.{ext?}`. When values for both `filename` and `ext` exist, both values are populated. If only a value for `filename` exists in the URL, the route matches because the trailing `.` is optional. The following URLs match this route:

- `/files/myFile.txt`
- `/files/myFile`

Route parameters may have **default values** designated by specifying the default value after the parameter name separated by an equals sign (`=`). For example, `{controller=Home}` defines `Home` as the default value for `controller`. The default value is used if no value is present in the URL for the parameter. Route parameters are made optional by appending a question mark (`?`) to the end of the parameter name. For example, `id?`. The difference between optional values and default route parameters is:

- A route parameter with a default value always produces a value.
- An optional parameter has a value only when a value is provided by the request URL.

Route parameters may have constraints that must match the route value bound from the URL. Adding `:` and constraint name after the route parameter name specifies an inline constraint on a route parameter. If the constraint requires arguments, they're enclosed in parentheses `(...)` after the constraint name. Multiple *inline constraints* can be specified by appending another `:` and constraint name.

The constraint name and arguments are passed to the [InlineConstraintResolver](#) service to create an instance of [IRouteConstraint](#) to use in URL processing. For example, the route template `blog/{article:minlength(10)}` specifies a `minlength` constraint with the argument `10`. For more information on route constraints and a list of the constraints provided by the framework, see the [Route constraints](#) section.

Route parameters may also have parameter transformers. Parameter transformers transform a parameter's value when generating links and matching actions and pages to URLs. Like constraints, parameter transformers can be added inline to a route parameter by adding a `:` and transformer name after the route parameter name. For example, the route template `blog/{article:slugify}` specifies a `slugify` transformer. For more information on parameter transformers, see the [Parameter transformers](#) section.

The following table demonstrates example route templates and their behavior:

[Expand table](#)

Route Template	Example Matching URI	The request URI...
<code>hello</code>	<code>/hello</code>	Only matches the single path <code>/hello</code> .
<code>{Page=Home}</code>	<code>/</code>	Matches and sets <code>Page</code> to <code>Home</code> .
<code>{Page=Home}</code>	<code>/Contact</code>	Matches and sets <code>Page</code> to <code>Contact</code> .
<code>{controller}/{action}/{id?}</code>	<code>/Products/List</code>	Maps to the <code>Products</code> controller and <code>List</code> action.
<code>{controller}/{action}/{id?}</code>	<code>/Products/Details/123</code>	Maps to the <code>Products</code> controller and <code>Details</code> action with <code>id</code> set to 123.
<code>{controller=Home}/{action=Index}/{id?}</code>	<code>/</code>	Maps to the <code>Home</code> controller and <code>Index</code> method. <code>id</code> is ignored.
<code>{controller=Home}/{action=Index}/{id?}</code>	<code>/Products</code>	Maps to the <code>Products</code> controller and <code>Index</code> method. <code>id</code> is ignored.

Using a template is generally the simplest approach to routing. Constraints and defaults can also be specified outside the route template.

## Complex segments

Complex segments are processed by matching up literal delimiters from right to left in a [non-greedy](#) way. For example, `[Route("/a{b}c{d}")]` is a complex segment. Complex segments work in a particular way that must be understood to use them successfully.

The example in this section demonstrates why complex segments only really work well when the delimiter text doesn't appear inside the parameter values. Using a [regex](#) and then manually extracting the values is needed for more complex cases.

### ⚠ Warning

When using `System.Text.RegularExpressions` to process untrusted input, pass a timeout. A malicious user can provide input to `Regex` causing a [Denial-of-Service attack](#). ASP.NET Core framework APIs that use `Regex` pass a timeout.

This is a summary of the steps that routing performs with the template `/a{b}c{d}` and the URL path `/abcd`. The `|` is used to help visualize how the algorithm works:

- The first literal, right to left, is `c`. So `/abcd` is searched from right and finds `/ab|c|d`.
- Everything to the right (`d`) is now matched to the route parameter `{d}`.
- The next literal, right to left, is `a`. So `/ab|c|d` is searched starting where we left off, then `a` is found `/|a|b|c|d`.
- The value to the right (`b`) is now matched to the route parameter `{b}`.
- There is no remaining text and no remaining route template, so this is a match.

Here's an example of a negative case using the same template `/a{b}c{d}` and the URL path `/aabcd`. The `|` is used to help visualize how the algorithm works. This case isn't a match, which is explained by the same algorithm:

- The first literal, right to left, is `c`. So `/aabcd` is searched from right and finds `/aab|c|d`.
- Everything to the right (`d`) is now matched to the route parameter `{d}`.
- The next literal, right to left, is `a`. So `/aab|c|d` is searched starting where we left off, then `a` is found `/a|a|b|c|d`.
- The value to the right (`b`) is now matched to the route parameter `{b}`.
- At this point there is remaining text `a`, but the algorithm has run out of route template to parse, so this is not a match.

Since the matching algorithm is [non-greedy](#):

- It matches the smallest amount of text possible in each step.
- Any case where the delimiter value appears inside the parameter values results in not matching.

Regular expressions provide much more control over their matching behavior.

Greedy matching, also known as *maximal matching* attempts to find the longest possible match in the input text that satisfies the [regex](#) pattern. Non-greedy matching, also known as *lazy matching*, seeks the shortest possible match in the input text that satisfies the regex pattern.

## Routing with special characters

Routing with special characters can lead to unexpected results. For example, consider a controller with the following action method:

C#

```
[HttpGet("{id?}/name")]
public async Task<ActionResult<string>> GetName(string id)
{
    var todoItem = await _context.TODOItems.FindAsync(id);

    if (todoItem == null || todoItem.Name == null)
    {
        return NotFound();
    }

    return todoItem.Name;
}
```

When `string id` contains the following encoded values, unexpected results might occur:

[Expand table](#)

ASCII	Encoded
/	%2F
+	+

Route parameters are not always URL decoded. This problem may be addressed in the future. For more information, see [this GitHub issue](#);

## Route constraints

Route constraints execute when a match has occurred to the incoming URL and the URL path is tokenized into route values. Route constraints generally inspect the route value

associated via the route template and make a true or false decision about whether the value is acceptable. Some route constraints use data outside the route value to consider whether the request can be routed. For example, the [HttpMethodRouteConstraint](#) can accept or reject a request based on its HTTP verb. Constraints are used in routing requests and link generation.

### Warning

Don't use constraints for input validation. If constraints are used for input validation, invalid input results in a `404` Not Found response. Invalid input should produce a `400` Bad Request with an appropriate error message. Route constraints are used to disambiguate similar routes, not to validate the inputs for a particular route.

The following table demonstrates example route constraints and their expected behavior:

 Expand table

constraint	Example	Example Matches	Notes
<code>int</code>	<code>{id:int}</code>	<code>123456789</code> , <code>-123456789</code>	Matches any integer
<code>bool</code>	<code>{active:bool}</code>	<code>true</code> , <code>FALSE</code>	Matches <code>true</code> or <code>false</code> . Case-insensitive
<code>datetime</code>	<code>{dob:datetime}</code>	<code>2016-12-31</code> , <code>2016-12-31</code> <code>7:32pm</code>	Matches a valid <code>DateTime</code> value in the invariant culture. See preceding warning.
<code>decimal</code>	<code>{price:decimal}</code>	<code>49.99</code> , <code>-1,000.01</code>	Matches a valid <code>decimal</code> value in the invariant culture. See preceding warning.
<code>double</code>	<code>{weight:double}</code>	<code>1.234</code> , <code>-1,001.01e8</code>	Matches a valid <code>double</code> value in the invariant culture. See preceding warning.
<code>float</code>	<code>{weight:float}</code>	<code>1.234</code> , <code>-1,001.01e8</code>	Matches a valid <code>float</code> value in the invariant



constraint	Example	Example Matches	Notes
			culture. See preceding warning.
guid	{id:guid}	CD2C1638-1638-72D5-1638-DEADBEEF1638	Matches a valid <code>Guid</code> value
long	{ticks:long}	123456789, -123456789	Matches a valid <code>long</code> value
minlength(value)	{username:minlength(4)}	Rick	String must be at least 4 characters
maxlength(value)	{filename:maxlength(8)}	MyFile	String must be no more than 8 characters
length(length)	{filename:length(12)}	somefile.txt	String must be exactly 12 characters long
length(min,max)	{filename:length(8,16)}	somefile.txt	String must be at least 8 and no more than 16 characters long
min(value)	{age:min(18)}	19	Integer value must be at least 18
max(value)	{age:max(120)}	91	Integer value must be no more than 120
range(min,max)	{age:range(18,120)}	91	Integer value must be at least 18 but no more than 120
alpha	{name:alpha}	Rick	String must consist of one or more alphabetical characters, <code>a-z</code> and case-insensitive.
regex(expression)	{ssn:regex(^\\d{3}-\\d{2}-\\d{4}\$)}	123-45-6789	String must match the regular expression. See tips about defining a regular expression.
required	{name:required}	Rick	Used to enforce that a non-parameter value is present during URL generation

### Warning

When using [System.Text.RegularExpressions](#) to process untrusted input, pass a timeout. A malicious user can provide input to `Regex` causing a [Denial-of-Service attack](#)<sup>↗</sup>. ASP.NET Core framework APIs that use `Regex` pass a timeout.

Multiple, colon delimited constraints can be applied to a single parameter. For example, the following constraint restricts a parameter to an integer value of 1 or greater:

C#

```
[Route("users/{id:int:min(1)}")]  
public User GetById(int id) { }
```

### Warning

Route constraints that verify the URL and are converted to a CLR type always use the invariant culture. For example, conversion to the CLR type `int` or `DateTime`. These constraints assume that the URL is not localizable. The framework-provided route constraints don't modify the values stored in route values. All route values parsed from the URL are stored as strings. For example, the `float` constraint attempts to convert the route value to a float, but the converted value is used only to verify it can be converted to a float.

## Regular expressions in constraints

### Warning

When using [System.Text.RegularExpressions](#) to process untrusted input, pass a timeout. A malicious user can provide input to `Regex` causing a [Denial-of-Service attack](#)<sup>↗</sup>. ASP.NET Core framework APIs that use `Regex` pass a timeout.

Regular expressions can be specified as inline constraints using the `regex(...)` route constraint. Methods in the `MapControllerRoute` family also accept an object literal of constraints. If that form is used, string values are interpreted as regular expressions.

The following code uses an inline regex constraint:

C#

```
app.MapGet("{message:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}",  
    () => "Inline Regex Constraint Matched");
```

The following code uses an object literal to specify a regex constraint:

C#

```
app.MapControllerRoute(  
    name: "people",  
    pattern: "people/{ssn}",  
    constraints: new { ssn = "^\\d{{3}}-\\d{{2}}-\\d{{4}}$", },  
    defaults: new { controller = "People", action = "List" });
```

The ASP.NET Core framework adds `RegexOptions.IgnoreCase` | `RegexOptions.Compiled` | `RegexOptions.CultureInvariant` to the regular expression constructor. See [RegexOptions](#) for a description of these members.

Regular expressions use delimiters and tokens similar to those used by routing and the C# language. Regular expression tokens must be escaped. To use the regular expression `^\\d{{3}}-\\d{{2}}-\\d{{4}}$` in an inline constraint, use one of the following:

- Replace `\\` characters provided in the string as `\\` characters in the C# source file in order to escape the `\\` string escape character.
- [Verbatim string literals](#).

To escape routing parameter delimiter characters `{`, `}`, `[`, `]`, double the characters in the expression, for example, `{{{`, `}}}`, `[[[`, `]]]`. The following table shows a regular expression and its escaped version:

[Expand table](#)

Regular expression	Escaped regular expression
<code>^\\d{{3}}-\\d{{2}}-\\d{{4}}\$</code>	<code>^\\d{{{3}}}-\\d{{2}}-\\d{{{4}}}\$</code>
<code>^[a-z]{{2}}\$</code>	<code>^[[[a-z]]{{{2}}}\$</code>

Regular expressions used in routing often start with the `^` character and match the starting position of the string. The expressions often end with the `$` character and match the end of the string. The `^` and `$` characters ensure that the regular expression matches the entire route parameter value. Without the `^` and `$` characters, the regular

expression matches any substring within the string, which is often undesirable. The following table provides examples and explains why they match or fail to match:

[Expand table](#)

Expression	String	Match	Comment
<code>[a-z]{2}</code>	hello	Yes	Substring matches
<code>[a-z]{2}</code>	123abc456	Yes	Substring matches
<code>[a-z]{2}</code>	mz	Yes	Matches expression
<code>[a-z]{2}</code>	MZ	Yes	Not case sensitive
<code>^[a-z]{2}\$</code>	hello	No	See <code>^</code> and <code>\$</code> above
<code>^[a-z]{2}\$</code>	123abc456	No	See <code>^</code> and <code>\$</code> above

For more information on regular expression syntax, see [.NET Framework Regular Expressions](#).

To constrain a parameter to a known set of possible values, use a regular expression. For example, `{action:regex:^(list|get|create)$}` only matches the `action` route value to `list`, `get`, or `create`. If passed into the constraints dictionary, the string `^(list|get|create)$` is equivalent. Constraints that are passed in the constraints dictionary that don't match one of the known constraints are also treated as regular expressions. Constraints that are passed within a template that don't match one of the known constraints are not treated as regular expressions.

## Custom route constraints

Custom route constraints can be created by implementing the [IRouteConstraint](#) interface. The `IRouteConstraint` interface contains [Match](#), which returns `true` if the constraint is satisfied and `false` otherwise.

Custom route constraints are rarely needed. Before implementing a custom route constraint, consider alternatives, such as model binding.

The ASP.NET Core [Constraints](#) folder provides good examples of creating constraints. For example, [GuidRouteConstraint](#).

To use a custom `IRouteConstraint`, the route constraint type must be registered with the app's [ConstraintMap](#) in the service container. A `ConstraintMap` is a dictionary that maps route constraint keys to `IRouteConstraint` implementations that validate those

constraints. An app's `ConstraintMap` can be updated in `Program.cs` either as part of an `AddRouting` call or by configuring `RouteOptions` directly with `builder.Services.Configure<RouteOptions>`. For example:

C#

```
builder.Services.AddRouting(options =>
    options.ConstraintMap.Add("noZeroes", typeof(NoZeroesRouteConstraint))));
```

The preceding constraint is applied in the following code:

C#

```
[ApiController]
[Route("api/[controller]")]
public class NoZeroesController : ControllerBase
{
    [HttpGet("{id:noZeroes}")]
    public IActionResult Get(string id) =>
        Content(id);
}
```

The implementation of `NoZeroesRouteConstraint` prevents `0` being used in a route parameter:

C#

```
public class NoZeroesRouteConstraint : IRouteConstraint
{
    private static readonly Regex _regex = new(
        @"^[1-9]*$",
        RegexOptions.CultureInvariant | RegexOptions.IgnoreCase,
        TimeSpan.FromMilliseconds(100));

    public bool Match(
        HttpContext? httpContext, IRouter? route, string routeKey,
        RouteValueDictionary values, RouteDirection routeDirection)
    {
        if (!values.TryGetValue(routeKey, out var routeValue))
        {
            return false;
        }

        var routeValueString = Convert.ToString(routeValue,
            CultureInfo.InvariantCulture);

        if (routeValueString is null)
        {
            return false;
        }
    }
}
```

```
        return _regex.IsMatch(routeValueString);
    }
}
```

### ⚠ Warning

When using [System.Text.RegularExpressions](#) to process untrusted input, pass a timeout. A malicious user can provide input to `Regex` causing a [Denial-of-Service attack](#). ASP.NET Core framework APIs that use `Regex` pass a timeout.

The preceding code:

- Prevents `0` in the `{id}` segment of the route.
- Is shown to provide a basic example of implementing a custom constraint. It should not be used in a production app.

The following code is a better approach to preventing an `id` containing a `0` from being processed:

C#

```
[HttpGet("{id}")]
public IActionResult Get(string id)
{
    if (id.Contains('0'))
    {
        return StatusCode(StatusCode.Status406NotAcceptable);
    }

    return Content(id);
}
```

The preceding code has the following advantages over the `NoZeroesRouteConstraint` approach:

- It doesn't require a custom constraint.
- It returns a more descriptive error when the route parameter includes `0`.

## Parameter transformers

Parameter transformers:

- Execute when generating a link using [LinkGenerator](#).
- Implement [Microsoft.AspNetCore.Routing.IOutboundParameterTransformer](#).
- Are configured using [ConstraintMap](#).
- Take the parameter's route value and transform it to a new string value.
- Result in using the transformed value in the generated link.

For example, a custom `slugify` parameter transformer in route pattern `blog\{article:slugify}` with `Url.Action(new { article = "MyTestArticle" })` generates `blog\my-test-article`.

Consider the following `IOutboundParameterTransformer` implementation:

C#

```
public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string? TransformOutbound(object? value)
    {
        if (value is null)
        {
            return null;
        }

        return Regex.Replace(
            value.ToString(),
            "([a-z])([A-Z])",
            "$1-$2",
            RegexOptions.CultureInvariant,
            TimeSpan.FromMilliseconds(100))
            .ToLowerInvariant();
    }
}
```

To use a parameter transformer in a route pattern, configure it using [ConstraintMap](#) in `Program.cs`:

C#

```
builder.Services.AddRouting(options =>
    options.ConstraintMap["slugify"] = typeof(SlugifyParameterTransformer));
```

The ASP.NET Core framework uses parameter transformers to transform the URI where an endpoint resolves. For example, parameter transformers transform the route values used to match an `area`, `controller`, `action`, and `page`:

C#

```
app.MapControllerRoute(  
    name: "default",  
    pattern: "{controller:slugify=Home}/{action:slugify=Index}/{id?}");
```

With the preceding route template, the action

`SubscriptionManagementController.GetAll` is matched with the URI `/subscription-management/get-all`. A parameter transformer doesn't change the route values used to generate a link. For example, `Url.Action("GetAll", "SubscriptionManagement")` outputs `/subscription-management/get-all`.

ASP.NET Core provides API conventions for using parameter transformers with generated routes:

- The [Microsoft.AspNetCore.Mvc.ApplicationModels.RouteTokenTransformerConvention](#) MVC convention applies a specified parameter transformer to all attribute routes in the app. The parameter transformer transforms attribute route tokens as they are replaced. For more information, see [Use a parameter transformer to customize token replacement](#).
- Razor Pages uses the [PageRouteTransformerConvention](#) API convention. This convention applies a specified parameter transformer to all automatically discovered Razor Pages. The parameter transformer transforms the folder and file name segments of Razor Pages routes. For more information, see [Use a parameter transformer to customize page routes](#).

## URL generation reference

This section contains a reference for the algorithm implemented by URL generation. In practice, most complex examples of URL generation use controllers or Razor Pages. See [routing in controllers](#) for additional information.

The URL generation process begins with a call to [LinkGenerator.GetPathByAddress](#) or a similar method. The method is provided with an address, a set of route values, and optionally information about the current request from `HttpContext`.

The first step is to use the address to resolve a set of candidate endpoints using an [EndpointAddressScheme<TAddress>](#) that matches the address's type.

Once the set of candidates is found by the address scheme, the endpoints are ordered and processed iteratively until a URL generation operation succeeds. URL generation does **not** check for ambiguities, the first result returned is the final result.



# Troubleshooting URL generation with logging

The first step in troubleshooting URL generation is setting the logging level of `Microsoft.AspNetCore.Routing` to `TRACE`. `LinkGenerator` logs many details about its processing which can be useful to troubleshoot problems.

See [URL generation reference](#) for details on URL generation.

## Addresses

Addresses are the concept in URL generation used to bind a call into the link generator to a set of candidate endpoints.

Addresses are an extensible concept that come with two implementations by default:

- Using *endpoint name* (`string`) as the address:
  - Provides similar functionality to MVC's route name.
  - Uses the `IEndpointNameMetadata` metadata type.
  - Resolves the provided string against the metadata of all registered endpoints.
  - Throws an exception on startup if multiple endpoints use the same name.
  - Recommended for general-purpose use outside of controllers and Razor Pages.
- Using *route values* (`RouteValuesAddress`) as the address:
  - Provides similar functionality to controllers and Razor Pages legacy URL generation.
  - Very complex to extend and debug.
  - Provides the implementation used by `IUrlHelper`, Tag Helpers, HTML Helpers, Action Results, etc.

The role of the address scheme is to make the association between the address and matching endpoints by arbitrary criteria:

- The endpoint name scheme performs a basic dictionary lookup.
- The route values scheme has a complex best subset of set algorithm.

## Ambient values and explicit values

From the current request, routing accesses the route values of the current request `HttpContext.Request.RouteValues`. The values associated with the current request are referred to as the **ambient values**. For the purpose of clarity, the documentation refers to the route values passed in to methods as **explicit values**.

The following example shows ambient values and explicit values. It provides ambient values from the current request and explicit values:

C#

```
public class WidgetController : ControllerBase
{
    private readonly LinkGenerator _linkGenerator;

    public WidgetController(LinkGenerator linkGenerator) =>
        _linkGenerator = linkGenerator;

    public IActionResult Index()
    {
        var indexPath = _linkGenerator.GetPathByAction(
            HttpContext, values: new { id = 17 }!);

        return Content(indexPath);
    }

    // ...
}
```

The preceding code:

- Returns `/Widget/Index/17`
- Gets [LinkGenerator](#) via [DI](#).

The following code provides only explicit values and no ambient values:

C#

```
var subscribePath = _linkGenerator.GetPathByAction(
    "Subscribe", "Home", new { id = 17 }!);
```

The preceding method returns `/Home/Subscribe/17`

The following code in the `WidgetController` returns `/Widget/Subscribe/17`:

C#

```
var subscribePath = _linkGenerator.GetPathByAction(
    HttpContext, "Subscribe", null, new { id = 17 });
```

The following code provides the controller from ambient values in the current request and explicit values:

C#

```
public class GadgetController : ControllerBase
{
    public IActionResult Index() =>
        Content(Url.Action("Edit", new { id = 17 }));
}
```

In the preceding code:

- `/Gadget/Edit/17` is returned.
- `Url` gets the `IUrlHelper`.
- `Action` generates a URL with an absolute path for an action method. The URL contains the specified `action` name and `route` values.

The following code provides ambient values from the current request and explicit values:

```
C#

public class IndexModel : PageModel
{
    public void OnGet()
    {
        var editUrl = Url.Page("./Edit", new { id = 17 });

        // ...
    }
}
```

The preceding code sets `url` to `/Edit/17` when the Edit Razor Page contains the following page directive:

```
@page "{id:int}"
```

If the Edit page doesn't contain the `"{id:int}"` route template, `url` is `/Edit?id=17`.

The behavior of MVC's `IUrlHelper` adds a layer of complexity in addition to the rules described here:

- `IUrlHelper` always provides the route values from the current request as ambient values.
- `IUrlHelper.Action` always copies the current `action` and `controller` route values as explicit values unless overridden by the developer.
- `IUrlHelper.Page` always copies the current `page` route value as an explicit value unless overridden.

- `UrlHelper.Page` always overrides the current `handler` route value with `null` as an explicit value unless overridden.

Users are often surprised by the behavioral details of ambient values, because MVC doesn't seem to follow its own rules. For historical and compatibility reasons, certain route values such as `action`, `controller`, `page`, and `handler` have their own special-case behavior.

The equivalent functionality provided by `LinkGenerator.GetPathByAction` and `LinkGenerator.GetPathByPage` duplicates these anomalies of `UrlHelper` for compatibility.

## URL generation process

Once the set of candidate endpoints are found, the URL generation algorithm:

- Processes the endpoints iteratively.
- Returns the first successful result.

The first step in this process is called **route value invalidation**. Route value invalidation is the process by which routing decides which route values from the ambient values should be used and which should be ignored. Each ambient value is considered and either combined with the explicit values, or ignored.

The best way to think about the role of ambient values is that they attempt to save application developers typing, in some common cases. Traditionally, the scenarios where ambient values are helpful are related to MVC:

- When linking to another action in the same controller, the controller name doesn't need to be specified.
- When linking to another controller in the same area, the area name doesn't need to be specified.
- When linking to the same action method, route values don't need to be specified.
- When linking to another part of the app, you don't want to carry over route values that have no meaning in that part of the app.

Calls to `LinkGenerator` or `UrlHelper` that return `null` are usually caused by not understanding route value invalidation. Troubleshoot route value invalidation by explicitly specifying more of the route values to see if that solves the problem.

Route value invalidation works on the assumption that the app's URL scheme is hierarchical, with a hierarchy formed from left-to-right. Consider the basic controller route template `{controller}/{action}/{id?}` to get an intuitive sense of how this works

in practice. A **change** to a value **invalidates** all of the route values that appear to the right. This reflects the assumption about hierarchy. If the app has an ambient value for `id`, and the operation specifies a different value for the `controller`:

- `id` won't be reused because `{controller}` is to the left of `{id?}`.

Some examples demonstrating this principle:

- If the explicit values contain a value for `id`, the ambient value for `id` is ignored. The ambient values for `controller` and `action` can be used.
- If the explicit values contain a value for `action`, any ambient value for `action` is ignored. The ambient values for `controller` can be used. If the explicit value for `action` is different from the ambient value for `action`, the `id` value won't be used. If the explicit value for `action` is the same as the ambient value for `action`, the `id` value can be used.
- If the explicit values contain a value for `controller`, any ambient value for `controller` is ignored. If the explicit value for `controller` is different from the ambient value for `controller`, the `action` and `id` values won't be used. If the explicit value for `controller` is the same as the ambient value for `controller`, the `action` and `id` values can be used.

This process is further complicated by the existence of attribute routes and dedicated conventional routes. Controller conventional routes such as `{controller}/{action}/{id?}` specify a hierarchy using route parameters. For [dedicated conventional routes](#) and [attribute routes](#) to controllers and Razor Pages:

- There is a hierarchy of route values.
- They don't appear in the template.

For these cases, URL generation defines the **required values** concept. Endpoints created by controllers and Razor Pages have required values specified that allow route value invalidation to work.

The route value invalidation algorithm in detail:

- The required value names are combined with the route parameters, then processed from left-to-right.
- For each parameter, the ambient value and explicit value are compared:
  - If the ambient value and explicit value are the same, the process continues.
  - If the ambient value is present and the explicit value isn't, the ambient value is used when generating the URL.

- If the ambient value isn't present and the explicit value is, reject the ambient value and all subsequent ambient values.
- If the ambient value and the explicit value are present, and the two values are different, reject the ambient value and all subsequent ambient values.

At this point, the URL generation operation is ready to evaluate route constraints. The set of accepted values is combined with the parameter default values, which is provided to constraints. If the constraints all pass, the operation continues.

Next, the **accepted values** can be used to expand the route template. The route template is processed:

- From left-to-right.
- Each parameter has its accepted value substituted.
- With the following special cases:
  - If the accepted values is missing a value and the parameter has a default value, the default value is used.
  - If the accepted values is missing a value and the parameter is optional, processing continues.
  - If any route parameter to the right of a missing optional parameter has a value, the operation fails.
  - Contiguous default-valued parameters and optional parameters are collapsed where possible.

Values explicitly provided that don't match a segment of the route are added to the query string. The following table shows the result when using the route template

`{controller}/{action}/{id?}`.

[Expand table](#)

Ambient Values	Explicit Values	Result
controller = "Home"	action = "About"	/Home/About
controller = "Home"	controller = "Order", action = "About"	/Order/About
controller = "Home", color = "Red"	action = "About"	/Home/About
controller = "Home"	action = "About", color = "Red"	/Home/About? color=Red

## Optional route parameter order

Optional route parameters must come after all required route parameters and literals. In the following code, the `id` and `name` parameters must come after the `color` parameter:

C#

```
using Microsoft.AspNetCore.Mvc;

namespace WebApplication1.Controllers;

[Route("api/[controller]")]
public class MyController : ControllerBase
{
    // GET /api/my/red/2/joe
    // GET /api/my/red/2
    // GET /api/my
    [HttpGet("{color}/{id:int?}/{name?}")]
    public IActionResult GetByIdAndOptionalName(string color, int id = 1,
        string? name = null)
    {
        return Ok($"{color} {id} {name ?? ""}");
    }
}
```

## Problems with route value invalidation

The following code shows an example of a URL generation scheme that's not supported by routing:

C#

```
app.MapControllerRoute(
    "default",
    "{culture}/{controller=Home}/{action=Index}/{id?}");

app.MapControllerRoute(
    "blog",
    "{culture}/{**slug}",
    new { controller = "Blog", action = "ReadPost" });
```

In the preceding code, the `culture` route parameter is used for localization. The desire is to have the `culture` parameter always accepted as an ambient value. However, the `culture` parameter is not accepted as an ambient value because of the way required values work:

- In the "default" route template, the `culture` route parameter is to the left of `controller`, so changes to `controller` won't invalidate `culture`.

- In the "blog" route template, the culture route parameter is considered to be to the right of controller, which appears in the required values.

## Parse URL paths with LinkParser

The `LinkParser` class adds support for parsing a URL path into a set of route values. The `ParsePathByEndpointName` method takes an endpoint name and a URL path, and returns a set of route values extracted from the URL path.

In the following example controller, the `GetProduct` action uses a route template of `api/Products/{id}` and has a `Name` of `GetProduct`:

C#

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet("{id}", Name = nameof(GetProduct))]
    public IActionResult GetProduct(string id)
    {
        // ...
    }
}
```

In the same controller class, the `AddRelatedProduct` action expects a URL path, `pathToRelatedProduct`, which can be provided as a query-string parameter:

C#

```
[HttpPost("{id}/Related")]
public IActionResult AddRelatedProduct(
    string id, string pathToRelatedProduct, [FromServices] LinkParser
    linkParser)
{
    var routeValues = linkParser.ParsePathByEndpointName(
        nameof(GetProduct), pathToRelatedProduct);
    var relatedProductId = routeValues?["id"];

    // ...
}
```

In the preceding example, the `AddRelatedProduct` action extracts the `id` route value from the URL path. For example, with a URL path of `/api/Products/1`, the `relatedProductId` value is set to `1`. This approach allows the API's clients to use URL paths when referring to resources, without requiring knowledge of how such a URL is structured.



# Configure endpoint metadata

The following links provide information on how to configure endpoint metadata:

- [Enable Cors with endpoint routing](#)
- [IAuthorizationPolicyProvider sample](#) [↗](#) using a custom `[MinimumAgeAuthorize]` attribute
- [Test authentication with the \[Authorize\] attribute](#)
- [RequireAuthorization](#)
- [Selecting the scheme with the \[Authorize\] attribute](#)
- [Apply policies using the \[Authorize\] attribute](#)
- [Role-based authorization in ASP.NET Core](#)

## Host matching in routes with RequireHost

`RequireHost` applies a constraint to the route which requires the specified host. The `RequireHost` or `[Host]` parameter can be a:

- Host: `www.domain.com`, matches `www.domain.com` with any port.
- Host with wildcard: `*.domain.com`, matches `www.domain.com`, `subdomain.domain.com`, or `www.subdomain.domain.com` on any port.
- Port: `*:5000`, matches port 5000 with any host.
- Host and port: `www.domain.com:5000` or `*.domain.com:5000`, matches host and port.

Multiple parameters can be specified using `RequireHost` or `[Host]`. The constraint matches hosts valid for any of the parameters. For example, `[Host("domain.com", "*.domain.com")]` matches `domain.com`, `www.domain.com`, and `subdomain.domain.com`.

The following code uses `RequireHost` to require the specified host on the route:

C#

```
app.MapGet("/", () => "Contoso").RequireHost("contoso.com");
app.MapGet("/", () => "AdventureWorks").RequireHost("adventure-works.com");

app.MapHealthChecks("/healthz").RequireHost("*:8080");
```

The following code uses the `[Host]` attribute on the controller to require any of the specified hosts:

C#

```
[Host("contoso.com", "adventure-works.com")]
public class HostsController : Controller
{
    public IActionResult Index() =>
        View();

    [Host("example.com")]
    public IActionResult Example() =>
        View();
}
```

When the `[Host]` attribute is applied to both the controller and action method:

- The attribute on the action is used.
- The controller attribute is ignored.

### Warning

API that relies on the [Host header](#), such as [HttpRequest.Host](#) and [RequireHost](#), are subject to potential spoofing by clients.

To prevent host and port spoofing, use one of the following approaches:

- Use [HttpContext.Connection](#) ([ConnectionInfo.LocalPort](#)) where the ports are checked.
- Employ [Host filtering](#).

## Route groups

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

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

C#

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

app.MapGroup("/private/todos")
    .MapTodosApi()
```

```

.WithTags("Private")
.AddEndpointFilterFactory(QueryPrivateTodos)
.RequireAuthorization();

EndpointFilterDelegate QueryPrivateTodos(EndpointFilterFactoryContext
factoryContext, EndpointFilterDelegate next)
{
    var dbContextIndex = -1;

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

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

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

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

```

C#

```

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

```

```
    return group;
}
```

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

C#

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

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

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

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

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

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

C#

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

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

C#

```

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

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

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

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

```

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

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

```
.NET CLI
```

```

/outer group filter
/inner group filter
MapGet filter

```

## Performance guidance for routing

When an app has performance problems, routing is often suspected as the problem. The reason routing is suspected is that frameworks like controllers and Razor Pages report the amount of time spent inside the framework in their logging messages. When there's a significant difference between the time reported by controllers and the total time of the request:

- Developers eliminate their app code as the source of the problem.
- It's common to assume routing is the cause.

Routing is performance tested using thousands of endpoints. It's unlikely that a typical app will encounter a performance problem just by being too large. The most common root cause of slow routing performance is usually a badly-behaving custom middleware.

This following code sample demonstrates a basic technique for narrowing down the source of delay:

C#

```
var logger = app.Services.GetRequiredService<ILogger<Program>>();

app.Use(async (context, next) =>
{
    var stopwatch = Stopwatch.StartNew();
    await next(context);
    stopwatch.Stop();

    logger.LogInformation("Time 1: {ElapsedMilliseconds}ms",
stopwatch.ElapsedMilliseconds);
});

app.UseRouting();

app.Use(async (context, next) =>
{
    var stopwatch = Stopwatch.StartNew();
    await next(context);
    stopwatch.Stop();

    logger.LogInformation("Time 2: {ElapsedMilliseconds}ms",
stopwatch.ElapsedMilliseconds);
});

app.UseAuthorization();

app.Use(async (context, next) =>
{
    var stopwatch = Stopwatch.StartNew();
    await next(context);
    stopwatch.Stop();

    logger.LogInformation("Time 3: {ElapsedMilliseconds}ms",
stopwatch.ElapsedMilliseconds);
});

app.MapGet("/", () => "Timing Test.");
```

To time routing:

- Interleave each middleware with a copy of the timing middleware shown in the preceding code.

- Add a unique identifier to correlate the timing data with the code.

This is a basic way to narrow down the delay when it's significant, for example, more than 10ms. Subtracting `Time 2` from `Time 1` reports the time spent inside the `UseRouting` middleware.

The following code uses a more compact approach to the preceding timing code:

C#

```
public sealed class AutoStopwatch : IDisposable
{
    private readonly ILogger _logger;
    private readonly string _message;
    private readonly Stopwatch _stopwatch;
    private bool _disposed;

    public AutoStopwatch(ILogger logger, string message) =>
        (_logger, _message, _stopwatch) = (logger, message,
        Stopwatch.StartNew());

    public void Dispose()
    {
        if (_disposed)
        {
            return;
        }

        _logger.LogInformation("{Message}: {ElapsedMilliseconds}ms",
            _message, _stopwatch.ElapsedMilliseconds);

        _disposed = true;
    }
}
```

C#

```
var logger = app.Services.GetRequiredService<ILogger<Program>>();
var timerCount = 0;

app.Use(async (context, next) =>
{
    using (new AutoStopwatch(logger, $"Time {++timerCount}"))
    {
        await next(context);
    }
});

app.UseRouting();

app.Use(async (context, next) =>
```

```

{
    using (new AutoStopwatch(logger, $"Time {++timerCount}"))
    {
        await next(context);
    }
});

app.UseAuthorization();

app.Use(async (context, next) =>
{
    using (new AutoStopwatch(logger, $"Time {++timerCount}"))
    {
        await next(context);
    }
});

app.MapGet("/", () => "Timing Test.");

```

## Potentially expensive routing features

The following list provides some insight into routing features that are relatively expensive compared with basic route templates:

- Regular expressions: It's possible to write regular expressions that are complex, or have long running time with a small amount of input.
- Complex segments (`{x}-{y}-{z}`):
  - Are significantly more expensive than parsing a regular URL path segment.
  - Result in many more substrings being allocated.
- Synchronous data access: Many complex apps have database access as part of their routing. Use extensibility points such as [MatcherPolicy](#) and [EndpointSelectorContext](#), which are asynchronous.

## Guidance for large route tables

By default ASP.NET Core uses a routing algorithm that trades memory for CPU time. This has the nice effect that route matching time is dependent only on the length of the path to match and not the number of routes. However, this approach can be potentially problematic in some cases, when the app has a large number of routes (in the thousands) and there is a high amount of variable prefixes in the routes. For example, if the routes have parameters in early segments of the route, like

```
{parameter}/some/literal.
```

It is unlikely for an app to run into a situation where this is a problem unless:



- There are a high number of routes in the app using this pattern.
- There is a large number of routes in the app.

## How to determine if an app is running into the large route table problem

- There are two symptoms to look for:
  - The app is slow to start on the first request.
    - Note that this is required but not sufficient. There are many other non-route problems than can cause slow app startup. Check for the condition below to accurately determine the app is running into this situation.
  - The app consumes a lot of memory during startup and a memory dump shows a large number of `Microsoft.AspNetCore.Routing.Matching.DfaNode` instances.

## How to address this issue

There are several techniques and optimizations that can be applied to routes that largely improve this scenario:

- Apply route constraints to your parameters, for example `{parameter:int}`, `{parameter:guid}`, `{parameter:regex(\\d+)}`, etc. where possible.
  - This allows the routing algorithm to internally optimize the structures used for matching and drastically reduce the memory used.
  - In the vast majority of cases this will suffice to get back to an acceptable behavior.
- Change the routes to move parameters to later segments in the template.
  - This reduces the number of possible "paths" to match an endpoint given a path.
- Use a dynamic route and perform the mapping to a controller/page dynamically.
  - This can be achieved using `MapDynamicControllerRoute` and `MapDynamicPageRoute`.

## Short-circuit middleware after routing

When routing matches an endpoint, it typically lets the rest of the middleware pipeline run before invoking the endpoint logic. Services can reduce resource usage by filtering out known requests early in the pipeline. Use the [ShortCircuit](#) extension method to cause routing to invoke the endpoint logic immediately and then end the request. For example, a given route might not need to go through authentication or CORS middleware. The following example short-circuits requests that match the `/short-circuit` route:

C#

```
app.MapGet("/short-circuit", () => "Short circuiting!").ShortCircuit();
```

The [ShortCircuit\(IEndpointConventionBuilder, Nullable<Int32>\)](#) method can optionally take a status code.

Use the [MapShortCircuit](#) method to set up short-circuiting for multiple routes at once, by passing to it a params array of URL prefixes. For example, browsers and bots often probe servers for well known paths like `robots.txt` and `favicon.ico`. If the app doesn't have those files, one line of code can configure both routes:

C#

```
app.MapShortCircuit(404, "robots.txt", "favicon.ico");
```

`MapShortCircuit` returns [IEndpointConventionBuilder](#) so that additional route constraints like host filtering can be added to it.

The `ShortCircuit` and `MapShortCircuit` methods do not affect middleware placed before `UseRouting`. Trying to use these methods with endpoints that also have `[Authorize]` or `[RequireCors]` metadata will cause requests to fail with an `InvalidOperationException`. This metadata is applied by [\[Authorize\]](#) or [\[EnableCors\]](#) attributes or by [RequireCors](#) or [RequireAuthorization](#) methods.

To see the effect of short-circuiting middleware, set the "Microsoft" logging category to "Information" in `appsettings.Development.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Information",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

Run the following code:

C#

```

var app = WebApplication.Create();

app.UseHttpLogging();

app.MapGet("/", () => "No short-circuiting!");
app.MapGet("/short-circuit", () => "Short circuiting!").ShortCircuit();
app.MapShortCircuit(404, "robots.txt", "favicon.ico");

app.Run();

```

The following example is from the console logs produced by running the `/` endpoint. It includes output from the logging middleware:

```

info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'HTTP: GET /'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'HTTP: GET /'
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 200
      Content-Type: text/plain; charset=utf-8
      Date: Wed, 03 May 2023 21:05:59 GMT
      Server: Kestrel
      Alt-Svc: h3=":5182"; ma=86400
      Transfer-Encoding: chunked

```

The following example is from running the `/short-circuit` endpoint. It doesn't have anything from the logging middleware because the middleware was short-circuited:

```

info: Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware[4]
      The endpoint 'HTTP: GET /short-circuit' is being executed without
      running additional middleware.
info: Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware[5]
      The endpoint 'HTTP: GET /short-circuit' has been executed without
      running additional middleware.

```

## Guidance for library authors

This section contains guidance for library authors building on top of routing. These details are intended to ensure that app developers have a good experience using libraries and frameworks that extend routing.

## Define endpoints

To create a framework that uses routing for URL matching, start by defining a user experience that builds on top of [UseEndpoints](#).

**DO** build on top of [IEndpointRouteBuilder](#). This allows users to compose your framework with other ASP.NET Core features without confusion. Every ASP.NET Core template includes routing. Assume routing is present and familiar for users.

C#

```
// Your framework
app.MapMyFramework(...);

app.MapHealthChecks("/healthz");
```

**DO** return a sealed concrete type from a call to `MapMyFramework(...)` that implements [IEndpointConventionBuilder](#). Most framework `Map...` methods follow this pattern. The `IEndpointConventionBuilder` interface:

- Allows for metadata to be composed.
- Is targeted by a variety of extension methods.

Declaring your own type allows you to add your own framework-specific functionality to the builder. It's ok to wrap a framework-declared builder and forward calls to it.

C#

```
// Your framework
app.MapMyFramework(...)
    .RequireAuthorization()
    .WithMyFrameworkFeature(awesome: true);

app.MapHealthChecks("/healthz");
```

**CONSIDER** writing your own [EndpointDataSource](#). `EndpointDataSource` is the low-level primitive for declaring and updating a collection of endpoints. `EndpointDataSource` is a powerful API used by controllers and Razor Pages. For more information, see [Dynamic endpoint routing](#) [↗](#).

The routing tests have a [basic example](#) [↗](#) of a non-updating data source.

**CONSIDER** implementing [GetGroupedEndpoints](#). This gives complete control over running group conventions and the final metadata on the grouped endpoints. For

example, this allows custom `EndpointDataSource` implementations to run [endpoint filters](#) added to groups.

**DO NOT** attempt to register an `EndpointDataSource` by default. Require users to register your framework in [UseEndpoints](#). The philosophy of routing is that nothing is included by default, and that `UseEndpoints` is the place to register endpoints.

## Creating routing-integrated middleware

**CONSIDER** defining metadata types as an interface.

**DO** make it possible to use metadata types as an attribute on classes and methods.

C#

```
public interface ICoolMetadata
{
    bool IsCool { get; }
}

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class CoolMetadataAttribute : Attribute, ICoolMetadata
{
    public bool IsCool => true;
}
```

Frameworks like controllers and Razor Pages support applying metadata attributes to types and methods. If you declare metadata types:

- Make them accessible as [attributes](#).
- Most users are familiar with applying attributes.

Declaring a metadata type as an interface adds another layer of flexibility:

- Interfaces are composable.
- Developers can declare their own types that combine multiple policies.

**DO** make it possible to override metadata, as shown in the following example:

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class SuppressCoolMetadataAttribute : Attribute, ICoolMetadata
{
    public bool IsCool => false;
}
```

```
[CoolMetadata]
public class MyController : Controller
{
    public void MyCool() { }

    [SuppressCoolMetadata]
    public void Uncool() { }
}
```

The best way to follow these guidelines is to avoid defining **marker metadata**:

- Don't just look for the presence of a metadata type.
- Define a property on the metadata and check the property.

The metadata collection is ordered and supports overriding by priority. In the case of controllers, metadata on the action method is most specific.

**DO** make middleware useful with and without routing:

C#

```
app.UseAuthorization(new AuthorizationPolicy() { ... });

// Your framework
app.MapMyFramework(...).RequireAuthorization();
```

As an example of this guideline, consider the `UseAuthorization` middleware. The authorization middleware allows you to pass in a fallback policy. The fallback policy, if specified, applies to both:

- Endpoints without a specified policy.
- Requests that don't match an endpoint.

This makes the authorization middleware useful outside of the context of routing. The authorization middleware can be used for traditional middleware programming.

## Debug diagnostics

For detailed routing diagnostic output, set `Logging:LogLevel:Microsoft` to `Debug`. In the development environment, set the log level in `appsettings.Development.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
```

```
"Default": "Information",  
"Microsoft": "Debug",  
"Microsoft.Hosting.Lifetime": "Information"  
}  
}  
}
```

## Additional resources

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

ⓘ **Note:** The author created this article with assistance from AI. [Learn more](#)

# Handle errors in ASP.NET Core

Article • 09/21/2024

## 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 [Tom Dykstra](#)

This article covers common approaches to handling errors in ASP.NET Core web apps. See also [Handle errors in ASP.NET Core controller-based web APIs](#) and [Handle errors in minimal APIs](#).

For Blazor error handling guidance, which adds to or supersedes the guidance in this article, see [Handle errors in ASP.NET Core Blazor apps](#).

## Developer exception page

The *Developer Exception Page* displays detailed information about unhandled request exceptions. It uses [DeveloperExceptionPageMiddleware](#) to capture synchronous and asynchronous exceptions from the HTTP pipeline and to generate error responses. The developer exception page runs early in the middleware pipeline, so that it can catch unhandled exceptions thrown in middleware that follows.

ASP.NET Core apps enable the developer exception page by default when both:

- Running in the [Development environment](#).
- The app was created with the current templates, that is, by using [WebApplication.CreateBuilder](#).

Apps created using earlier templates, that is, by using [WebHost.CreateDefaultBuilder](#), can enable the developer exception page by calling [app.UseDeveloperExceptionPage](#).

## Warning

Don't enable the Developer Exception Page **unless the app is running in the Development environment**. Don't share detailed exception information publicly



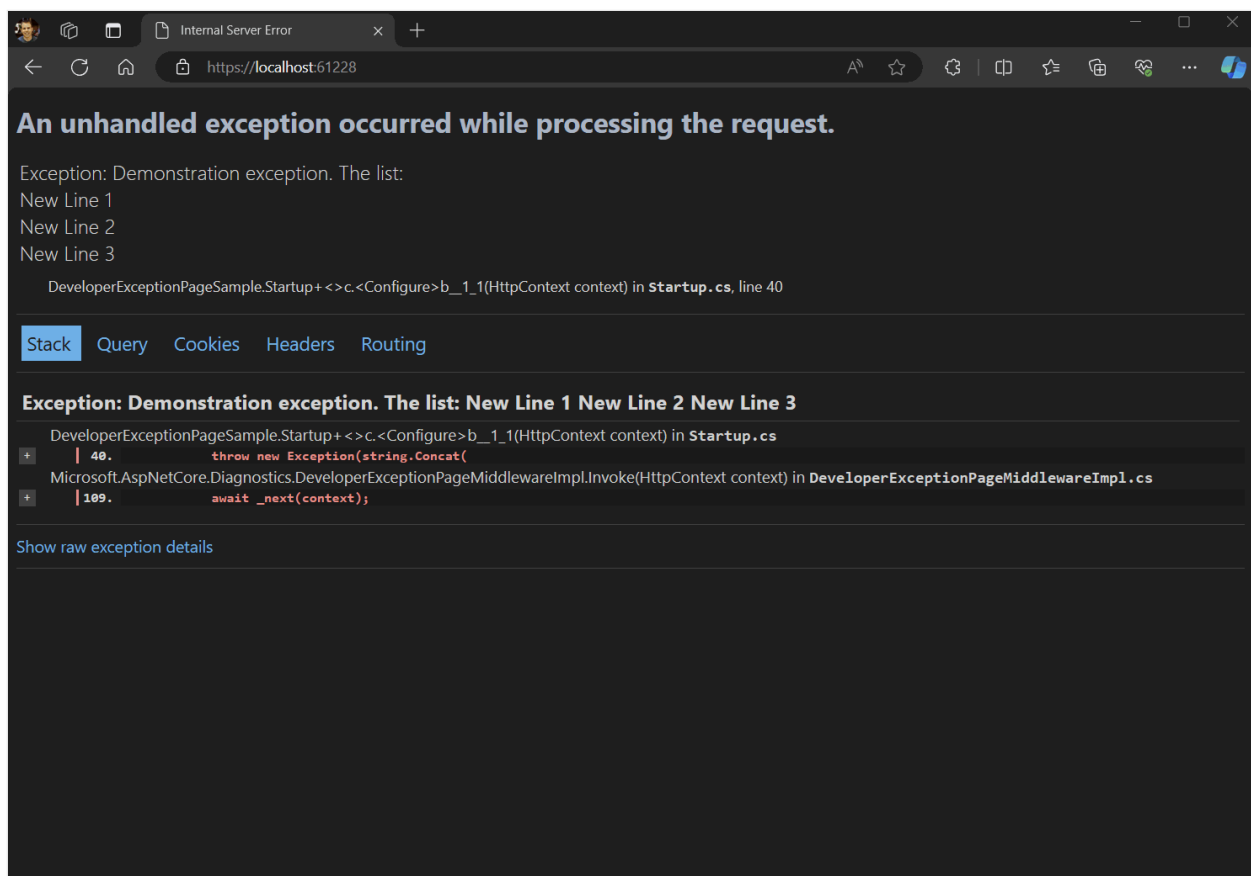
when the app runs in production. For more information on configuring environments, see [Use multiple environments in ASP.NET Core](#).

The Developer Exception Page can include the following information about the exception and the request:

- Stack trace
- Query string parameters, if any
- Cookies, if any
- Headers
- Endpoint metadata, if any

The Developer Exception Page isn't guaranteed to provide any information. Use [Logging](#) for complete error information.

The following image shows a sample developer exception page with animation to show the tabs and the information displayed:



In response to a request with an `Accept: text/plain` header, the Developer Exception Page returns plain text instead of HTML. For example:

text

Status: 500 Internal Server Error  
Time: 9.39 msSize: 480 bytes

```
FormattedRawHeadersRequest
Body
text/plain; charset=utf-8, 480 bytes
System.InvalidOperationException: Sample Exception
  at WebApplicationMinimal.Program.<>c.<Main>b__0_0() in
C:\Source\WebApplicationMinimal\Program.cs:line 12
  at lambda_method1(Closure, Object, HttpContext)
  at
Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddlewareImpl.Invoke
(HttpContext context)

HEADERS
=====
Accept: text/plain
Host: localhost:7267
traceparent: 00-0eab195ea19d07b90a46cd7d6bf2f
```

## Exception handler page

To configure a custom error handling page for the [Production environment](#), call [UseExceptionHandler](#). This exception handling middleware:

- Catches and logs unhandled exceptions.
- Re-executes the request in an alternate pipeline using the path indicated. The request isn't re-executed if the response has started. The template-generated code re-executes the request using the `/Error` path.

### Warning

If the alternate pipeline throws an exception of its own, Exception Handling Middleware rethrows the original exception.

Since this middleware can re-execute the request pipeline:

- Middlewares need to handle reentrancy with the same request. This normally means either cleaning up their state after calling `_next` or caching their processing on the `HttpContext` to avoid redoing it. When dealing with the request body, this either means buffering or caching the results like the Form reader.
- For the [UseExceptionHandler\(IApplicationBuilder, String\)](#) overload that is used in templates, only the request path is modified, and the route data is cleared. Request data such as headers, method, and items are all reused as-is.
- Scoped services remain the same.

In the following example, [UseExceptionHandler](#) adds the exception handling middleware in non-Development environments:

C#

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

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

The Razor Pages app template provides an Error page (`.cshtml`) and [PageModel](#) class (`ErrorModel`) in the *Pages* folder. For an MVC app, the project template includes an `Error` action method and an Error view for the Home controller.

The exception handling middleware re-executes the request using the *original* HTTP method. If an error handler endpoint is restricted to a specific set of HTTP methods, it runs only for those HTTP methods. For example, an MVC controller action that uses the `[HttpGet]` attribute runs only for GET requests. To ensure that *all* requests reach the custom error handling page, don't restrict them to a specific set of HTTP methods.

To handle exceptions differently based on the original HTTP method:

- For Razor Pages, create multiple handler methods. For example, use `OnGet` to handle GET exceptions and use `OnPost` to handle POST exceptions.
- For MVC, apply HTTP verb attributes to multiple actions. For example, use `[HttpGet]` to handle GET exceptions and use `[HttpPost]` to handle POST exceptions.

To allow unauthenticated users to view the custom error handling page, ensure that it supports anonymous access.

## Access the exception

Use [ExceptionHandlerPathFeature](#) to access the exception and the original request path in an error handler. The following example uses `ExceptionHandlerPathFeature` to get more information about the exception that was thrown:

C#

```
[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
```

```
[IgnoreAntiforgeryToken]
public class ErrorModel : PageModel
{
    public string? RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public string? ExceptionMessage { get; set; }

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;

        var exceptionHandlerPathFeature =
            HttpContext.Features.Get<IExceptionHandlerPathFeature>();

        if (exceptionHandlerPathFeature?.Error is FileNotFoundException)
        {
            ExceptionMessage = "The file was not found.";
        }

        if (exceptionHandlerPathFeature?.Path == "/")
        {
            ExceptionMessage ??= string.Empty;
            ExceptionMessage += " Page: Home.";
        }
    }
}
```

### Warning

Do **not** serve sensitive error information to clients. Serving errors is a security risk.

## Exception handler lambda

An alternative to a [custom exception handler page](#) is to provide a lambda to [UseExceptionHandler](#). Using a lambda allows access to the error before returning the response.

The following code uses a lambda for exception handling:

```
C#

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
```

```

app.UseExceptionHandler(exceptionHandlerApp =>
{
    exceptionHandlerApp.Run(async context =>
    {
        context.Response.StatusCode =
StatusCodes.Status500InternalServerError;

        // using static System.Net.Mime.MediaTypeNames;
        context.Response.ContentType = Text.Plain;

        await context.Response.WriteAsync("An exception was thrown.");

        var exceptionHandlerPathFeature =
            context.Features.Get<IExceptionHandlerPathFeature>();

        if (exceptionHandlerPathFeature?.Error is FileNotFoundException)
        {
            await context.Response.WriteAsync(" The file was not
found.");
        }

        if (exceptionHandlerPathFeature?.Path == "/")
        {
            await context.Response.WriteAsync(" Page: Home.");
        }
    });
});

app.UseHsts();
}

```

Another way to use a lambda is to set the status code based on the exception type, as in the following example:

```

C#

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails();
var app = builder.Build();
if (app.Environment.IsDevelopment())
{
    app.UseExceptionHandler(new ExceptionHandlerOptions
    {
        StatusCodeSelector = ex => ex is TimeoutException
            ? StatusCodes.Status503ServiceUnavailable
            : StatusCodes.Status500InternalServerError
    });
}

```

 **Warning**

Do **not** serve sensitive error information to clients. Serving errors is a security risk.

# ExceptionHandler

[ExceptionHandler](#) is an interface that gives the developer a callback for handling known exceptions in a central location.

`ExceptionHandler` implementations are registered by calling [IServiceCollection.AddExceptionHandler<T>](#). The lifetime of an `ExceptionHandler` instance is singleton. Multiple implementations can be added, and they're called in the order registered.

If an exception handler handles a request, it can return `true` to stop processing. If an exception isn't handled by any exception handler, then control falls back to the default behavior and options from the middleware. Different metrics and logs are emitted for handled versus unhandled exceptions.

The following example shows an `ExceptionHandler` implementation:

C#

```
using Microsoft.AspNetCore.Diagnostics;

namespace ErrorHandlingSample
{
    public class CustomExceptionHandler : IExceptionHandler
    {
        private readonly ILogger<CustomExceptionHandler> logger;
        public CustomExceptionHandler(ILogger<CustomExceptionHandler>
logger)
        {
            this.logger = logger;
        }
        public ValueTask<bool> TryHandleAsync(
            HttpContext httpContext,
            Exception exception,
            CancellationToken cancellationToken)
        {
            var exceptionMessage = exception.Message;
            logger.LogError(
                "Error Message: {exceptionMessage}, Time of occurrence
{time}",
                exceptionMessage, DateTime.UtcNow);
            // Return false to continue with the default behavior
            // - or - return true to signal that this exception is handled
            return ValueTask.FromResult(false);
        }
    }
}
```

```
}  
}
```

The following example shows how to register an `ExceptionHandler` implementation for dependency injection:

C#

```
using ErrorHandlingSample;  
  
var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddDatabaseDeveloperPageExceptionFilter();  
builder.Services.AddRazorPages();  
builder.Services.AddExceptionHandler<CustomExceptionHandler>();  
  
var app = builder.Build();  
  
if (!app.Environment.IsDevelopment())  
{  
    app.UseExceptionHandler("/Error");  
    app.UseHsts();  
}  
  
// Remaining Program.cs code omitted for brevity
```

When the preceding code runs in the Development environment:

- The `CustomExceptionHandler` is called first to handle an exception.
- After logging the exception, the `TryHandleAsync` method returns `false`, so the [developer exception page](#) is shown.

In other environments:

- The `CustomExceptionHandler` is called first to handle an exception.
- After logging the exception, the `TryHandleAsync` method returns `false`, so the [/Error page](#) is shown.

## UseStatusCodePages

By default, an ASP.NET Core app doesn't provide a status code page for HTTP error status codes, such as *404 - Not Found*. When the app sets an HTTP 400-599 error status code that doesn't have a body, it returns the status code and an empty response body. To enable default text-only handlers for common error status codes, call [UseStatusCodePages](#) in `Program.cs`:

C#

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

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

app.UseStatusCodePages();
```

Call `UseStatusCodePages` before request handling middleware. For example, call `UseStatusCodePages` before the Static File Middleware and the Endpoints Middleware.

When `UseStatusCodePages` isn't used, navigating to a URL without an endpoint returns a browser-dependent error message indicating the endpoint can't be found. When `UseStatusCodePages` is called, the browser returns the following response:

Console

Status Code: 404; Not Found

`UseStatusCodePages` isn't typically used in production because it returns a message that isn't useful to users.

### ⚠ Note

The status code pages middleware does **not** catch exceptions. To provide a custom error handling page, use the [exception handler page](#).

## UseStatusCodePages with format string

To customize the response content type and text, use the overload of `UseStatusCodePages` that takes a content type and format string:

C#

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

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



```
// using static System.Net.Mime.MediaTypeNames;  
app.UseStatusCodePages(Text.Plain, "Status Code Page: {0}");
```

In the preceding code, `{0}` is a placeholder for the error code.

`UseStatusCodePages` with a format string isn't typically used in production because it returns a message that isn't useful to users.

## UseStatusCodePages with lambda

To specify custom error-handling and response-writing code, use the overload of `UseStatusCodePages` that takes a lambda expression:

C#

```
var app = builder.Build();  
  
if (!app.Environment.IsDevelopment())  
{  
    app.UseExceptionHandler("/Error");  
    app.UseHsts();  
}  
  
app.UseStatusCodePages(async statusCodeContext =>  
{  
    // using static System.Net.Mime.MediaTypeNames;  
    statusCodeContext.HttpContext.Response.ContentType = Text.Plain;  
  
    await statusCodeContext.HttpContext.Response.WriteAsync(  
        $"Status Code Page:  
{statusCodeContext.HttpContext.Response.StatusCode}");  
});
```

`UseStatusCodePages` with a lambda isn't typically used in production because it returns a message that isn't useful to users.

## UseStatusCodePagesWithRedirects

The `UseStatusCodePagesWithRedirects` extension method:

- Sends a [302 - Found](#) status code to the client.
- Redirects the client to the error handling endpoint provided in the URL template. The error handling endpoint typically displays error information and returns HTTP 200.

C#

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

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

app.UseStatusCodePagesWithRedirects("/StatusCode/{0}");
```

The URL template can include a `{0}` placeholder for the status code, as shown in the preceding code. If the URL template starts with `~` (tilde), the `~` is replaced by the app's `PathBase`. When specifying an endpoint in the app, create an MVC view or Razor page for the endpoint.

This method is commonly used when the app:

- Should redirect the client to a different endpoint, usually in cases where a different app processes the error. For web apps, the client's browser address bar reflects the redirected endpoint.
- Shouldn't preserve and return the original status code with the initial redirect response.

## UseStatusCodePagesWithReExecute

The [UseStatusCodePagesWithReExecute](#) extension method:

- Generates the response body by re-executing the request pipeline using an alternate path.
- Does not alter the status code before or after re-executing the pipeline.

The new pipeline execution may alter the response's status code, as the new pipeline has full control of the status code. If the new pipeline does not alter the status code, the original status code will be sent to the client.

C#

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

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

```
app.UseStatusCodePagesWithReExecute("/StatusCode/{0}");
```

If an endpoint within the app is specified, create an MVC view or Razor page for the endpoint.

This method is commonly used when the app should:

- Process the request without redirecting to a different endpoint. For web apps, the client's browser address bar reflects the originally requested endpoint.
- Preserve and return the original status code with the response.

The URL template must start with `/` and may include a placeholder `{0}` for the status code. To pass the status code as a query-string parameter, pass a second argument into `UseStatusCodePagesWithReExecute`. For example:

C#

```
var app = builder.Build();  
app.UseStatusCodePagesWithReExecute("/StatusCode", "?statusCode={0}");
```

The endpoint that processes the error can get the original URL that generated the error, as shown in the following example:

C#

```
[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]  
public class StatusCodeModel : PageModel  
{  
    public int OriginalStatusCode { get; set; }  
  
    public string? OriginalPathAndQuery { get; set; }  
  
    public void OnGet(int statusCode)  
    {  
        OriginalStatusCode = statusCode;  
    }  
}
```

```

var statusCodeReExecuteFeature =
    HttpContext.Features.Get<IStatusCodeReExecuteFeature>();

if (statusCodeReExecuteFeature is not null)
{
    OriginalPathAndQuery = $"
{statusCodeReExecuteFeature.OriginalPathBase}"
                        + $"
{statusCodeReExecuteFeature.OriginalPath}"
                        + $"
{statusCodeReExecuteFeature.OriginalQueryString}";
}
}
}

```

Since this middleware can re-execute the request pipeline:

- Middlewares need to handle reentrancy with the same request. This normally means either cleaning up their state after calling `_next` or caching their processing on the `HttpContext` to avoid redoing it. When dealing with the request body, this either means buffering or caching the results like the Form reader.
- Scoped services remain the same.

## Disable status code pages

To disable status code pages for an MVC controller or action method, use the [\[SkipStatusCodePages\]](#) attribute.

To disable status code pages for specific requests in a Razor Pages handler method or in an MVC controller, use [IStatusCodePagesFeature](#):

```

C#

public void OnGet()
{
    var statusCodePagesFeature =
        HttpContext.Features.Get<IStatusCodePagesFeature>();

    if (statusCodePagesFeature is not null)
    {
        statusCodePagesFeature.Enabled = false;
    }
}

```

## Exception-handling code

Code in exception handling pages can also throw exceptions. Production error pages should be tested thoroughly and take extra care to avoid throwing exceptions of their own.

## Response headers

Once the headers for a response are sent:

- The app can't change the response's status code.
- Any exception pages or handlers can't run. The response must be completed or the connection aborted.

## Server exception handling

In addition to the exception handling logic in an app, the [HTTP server implementation](#) can handle some exceptions. If the server catches an exception before response headers are sent, the server sends a `500 - Internal Server Error` response without a response body. If the server catches an exception after response headers are sent, the server closes the connection. Requests that aren't handled by the app are handled by the server. Any exception that occurs when the server is handling the request is handled by the server's exception handling. The app's custom error pages, exception handling middleware, and filters don't affect this behavior.

## Startup exception handling

Only the hosting layer can handle exceptions that take place during app startup. The host can be configured to [capture startup errors](#) and [capture detailed errors](#).

The hosting layer can show an error page for a captured startup error only if the error occurs after host address/port binding. If binding fails:

- The hosting layer logs a critical exception.
- The dotnet process crashes.
- No error page is displayed when the HTTP server is [Kestrel](#).

When running on [IIS](#) (or Azure App Service) or [IIS Express](#), a `502.5 - Process Failure` is returned by the [ASP.NET Core Module](#) if the process can't start. For more information, see [Troubleshoot ASP.NET Core on Azure App Service and IIS](#).

## Database error page

The Database developer page exception filter

[AddDatabaseDeveloperPageExceptionFilter](#) captures database-related exceptions that can be resolved by using Entity Framework Core migrations. When these exceptions occur, an HTML response is generated with details of possible actions to resolve the issue. This page is enabled only in the Development environment. The following code adds the Database developer page exception filter:

C#

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

builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddRazorPages();
```

## Exception filters

In MVC apps, exception filters can be configured globally or on a per-controller or per-action basis. In Razor Pages apps, they can be configured globally or per page model. These filters handle any unhandled exceptions that occur during the execution of a controller action or another filter. For more information, see [Filters in ASP.NET Core](#).

Exception filters are useful for trapping exceptions that occur within MVC actions, but they're not as flexible as the built-in [exception handling middleware](#) [↗](#), [UseExceptionHandler](#). We recommend using `ExceptionHandler`, unless you need to perform error handling differently based on which MVC action is chosen.

## Model state errors

For information about how to handle model state errors, see [Model binding](#) and [Model validation](#).

## Problem details

[Problem Details](#) [↗](#) are not the only response format to describe an HTTP API error, however, they are commonly used to report errors for HTTP APIs.

The problem details service implements the [IProblemDetailsService](#) interface, which supports creating problem details in ASP.NET Core. The [AddProblemDetails\(IServiceCollection\)](#) extension method on [IServiceCollection](#) registers the default `IProblemDetailsService` implementation.

In ASP.NET Core apps, the following middleware generates problem details HTTP responses when `AddProblemDetails` is called, except when the [Accept request HTTP header](#) [↗](#) doesn't include one of the content types supported by the registered [IProblemDetailsWriter](#) (default: `application/json`):

- [ExceptionHandlerMiddleware](#): Generates a problem details response when a custom handler is not defined.
- [StatusCodePagesMiddleware](#): Generates a problem details response by default.
- [DeveloperExceptionPageMiddleware](#): Generates a problem details response in development when the `Accept` request HTTP header doesn't include `text/html`.

The following code configures the app to generate a problem details response for all HTTP client and server error responses that ***don't have body content yet***:

```
C#

builder.Services.AddProblemDetails();

var app = builder.Build();

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

app.UseStatusCodePages();
```

The next section shows how to customize the problem details response body.

## Customize problem details

The automatic creation of a `ProblemDetails` can be customized using any of the following options:

1. Use [ProblemDetailsOptions.CustomizeProblemDetails](#)
2. Use a custom [IProblemDetailsWriter](#)
3. Call the [IProblemDetailsService](#) in a middleware

### `CustomizeProblemDetails` operation

The generated problem details can be customized using [CustomizeProblemDetails](#), and the customizations are applied to all auto-generated problem details.

The following code uses [ProblemDetailsOptions](#) to set [CustomizeProblemDetails](#):

C#

```
builder.Services.AddProblemDetails(options =>
    options.CustomizeProblemDetails = ctx =>
        ctx.ProblemDetails.Extensions.Add("nodeId",
Environment.MachineName));

var app = builder.Build();

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

app.UseStatusCodePages();
```

For example, an [HTTP Status 400 Bad Request](#) endpoint result produces the following problem details response body:

JSON

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "Bad Request",
  "status": 400,
  "nodeId": "my-machine-name"
}
```

## Custom `IProblemDetailsWriter`

An `IProblemDetailsWriter` implementation can be created for advanced customizations.

C#

```
public class SampleProblemDetailsWriter : IProblemDetailsWriter
{
    // Indicates that only responses with StatusCode == 400
    // are handled by this writer. All others are
    // handled by different registered writers if available.
    public bool CanWrite(ProblemDetailsContext context)
        => context.HttpContext.Response.StatusCode == 400;

    public ValueTask WriteAsync(ProblemDetailsContext context)
    {
        // Additional customizations.

        // Write to the response.
        var response = context.HttpContext.Response;
```



```

        return new
        ValueTask(response.WriteAsJsonAsync(context.ProblemDetails));
    }
}

```

**Note:** When using a custom `IProblemDetailsWriter`, the custom `IProblemDetailsWriter` must be registered before calling [AddRazorPages](#), [AddControllers](#), [AddControllersWithViews](#), or [AddMvc](#):

C#

```

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddTransient<IProblemDetailsWriter,
SampleProblemDetailsWriter>();

var app = builder.Build();

// Middleware to handle writing problem details to the response.
app.Use(async (context, next) =>
{
    await next(context);
    var mathErrorFeature = context.Features.Get<MathErrorFeature>();
    if (mathErrorFeature is not null)
    {
        if (context.RequestServices.GetService<IProblemDetailsWriter>() is
            { } problemDetailsService)
        {
            if (problemDetailsService.CanWrite(new ProblemDetailsContext() {
HttpContext = context }))
            {
                (string Detail, string Type) details =
mathErrorFeature.MathError switch
                {
                    MathErrorType.DivisionByZeroError => ("Divison by zero
is not defined.",
                    "https://en.wikipedia.org/wiki/Division_by_zero"),
                    _ => ("Negative or complex numbers are not valid
input.",
                    "https://en.wikipedia.org/wiki/Square_root")
                };

                await problemDetailsService.WriteAsync(new
ProblemDetailsContext
                {
                    HttpContext = context,
                    ProblemDetails =
                    {
                        Title = "Bad Input",
                        Detail = details.Detail,
                        Type = details.Type
                    }
                }
            );
        }
    }
}
);

```

```

    }
    });
}
}
});

// /divide?numerator=2&denominator=4
app.MapGet("/divide", (HttpContext context, double numerator, double
denominator) =>
{
    if (denominator == 0)
    {
        var errorType = new MathErrorFeature
        {
            MathError = MathErrorType.DivisionByZeroError
        };
        context.Features.Set(errorType);
        return Results.BadRequest();
    }

    return Results.Ok(numerator / denominator);
});

// /squareroot?radicand=16
app.MapGet("/squareroot", (HttpContext context, double radicand) =>
{
    if (radicand < 0)
    {
        var errorType = new MathErrorFeature
        {
            MathError = MathErrorType.NegativeRadicandError
        };
        context.Features.Set(errorType);
        return Results.BadRequest();
    }

    return Results.Ok(Math.Sqrt(radicand));
});

app.Run();

```

## Problem details from Middleware

An alternative approach to using [ProblemDetailsOptions](#) with [CustomizeProblemDetails](#) is to set the [ProblemDetails](#) in middleware. A problem details response can be written by calling [IProblemDetailsService.WriteAsync](#):

C#

```

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStatusCodePages();

// Middleware to handle writing problem details to the response.
app.Use(async (context, next) =>
{
    await next(context);
    var mathErrorFeature = context.Features.Get<MathErrorFeature>();
    if (mathErrorFeature is not null)
    {
        if (context.RequestServices.GetService<IProblemDetailsService>() is
            problemDetailsService)
        {
            (string Detail, string Type) details =
            mathErrorFeature.MathError switch
            {
                MathErrorType.DivisionByZeroError => ("Divison by zero is
not defined.",
                "https://en.wikipedia.org/wiki/Division_by_zero"),
                _ => ("Negative or complex numbers are not valid input.",
                "https://en.wikipedia.org/wiki/Square_root")
            };

            await problemDetailsService.WriteAsync(new ProblemDetailsContext
            {
                HttpContext = context,
                ProblemDetails =
                {
                    Title = "Bad Input",
                    Detail = details.Detail,
                    Type = details.Type
                }
            });
        }
    }
});

// /divide?numerator=2&denominator=4
app.MapGet("/divide", (HttpContext context, double numerator, double
denominator) =>
{
    if (denominator == 0)
    {
        var errorType = new MathErrorFeature { MathError =

```

```

MathErrorType.DivisionByZeroError };
    context.Features.Set(errorType);
    return Results.BadRequest();
}

    return Results.Ok(numerator / denominator);
});

// /squareroot?radicand=16
app.MapGet("/squareroot", (HttpContext context, double radicand) =>
{
    if (radicand < 0)
    {
        var errorType = new MathErrorFeature { MathError =

MathErrorType.NegativeRadicandError };
        context.Features.Set(errorType);
        return Results.BadRequest();
    }

    return Results.Ok(Math.Sqrt(radicand));
});

app.MapControllers();

app.Run();

```

In the preceding code, the minimal API endpoints `/divide` and `/squareroot` return the expected custom problem response on error input.

The API controller endpoints return the default problem response on error input, not the custom problem response. The default problem response is returned because the API controller has written to the response stream, [Problem details for error status codes](#), before [IProblemDetailsService.WriteAsync](#) is called and the response is **not** written again.

The following `ValuesController` returns [BadRequestResult](#), which writes to the response stream and therefore prevents the custom problem response from being returned.

C#

```

[Route("api/[controller]/[action]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // /api/values/divide/1/2
    [HttpGet("{Numerator}/{Denominator}")]
    public IActionResult Divide(double Numerator, double Denominator)
    {

```

```

        if (Denominator == 0)
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.DivisionByZeroError
            };
            HttpContext.Features.Set(errorType);
            return BadRequest();
        }

        return Ok(Numerator / Denominator);
    }

    // /api/values/squareroot/4
    [HttpGet("{radicand}")]
    public IActionResult Squareroot(double radicand)
    {
        if (radicand < 0)
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.NegativeRadicandError
            };
            HttpContext.Features.Set(errorType);
            return BadRequest();
        }
        return Ok(Math.Sqrt(radicand));
    }
}

```

The following `Values3Controller` returns `ControllerBase.Problem` so the expected custom problem result is returned:

C#

```

[Route("api/[controller]/[action]")]
[ApiController]
public class Values3Controller : ControllerBase
{
    // /api/values3/divide/1/2
    [HttpGet("{Numerator}/{Denominator}")]
    public IActionResult Divide(double Numerator, double Denominator)
    {
        if (Denominator == 0)
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.DivisionByZeroError
            };
            HttpContext.Features.Set(errorType);

```

```

        return Problem(
            title: "Bad Input",
            detail: "Divison by zero is not defined.",
            type: "https://en.wikipedia.org/wiki/Division_by_zero",
            statusCode: StatusCodes.Status400BadRequest
        );
    }

    return Ok(Numerator / Denominator);
}

// /api/values3/squareroot/4
[HttpGet("{radicand}")]
public IActionResult Squareroot(double radicand)
{
    if (radicand < 0)
    {
        var errorType = new MathErrorFeature
        {
            MathError = MathErrorType.NegativeRadicandError
        };
        HttpContext.Features.Set(errorType);
        return Problem(
            title: "Bad Input",
            detail: "Negative or complex numbers are not valid input.",
            type: "https://en.wikipedia.org/wiki/Square_root",
            statusCode: StatusCodes.Status400BadRequest
        );
    }

    return Ok(Math.Sqrt(radicand));
}
}

```

## Produce a ProblemDetails payload for exceptions

Consider the following app:

C#

```

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseExceptionHandler();
app.UseStatusCodePages();

```

```

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

app.MapControllers();
app.Run();

```

In non-development environments, when an exception occurs, the following is a standard [ProblemDetails response](#) that is returned to the client:

JSON

```

{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.6.1",
  "title": "An error occurred while processing your request.",
  "status": 500, "traceId": "00-b644<snip>-00"
}

```

For most apps, the preceding code is all that's needed for exceptions. However, the following section shows how to get more detailed problem responses.

An alternative to a [custom exception handler page](#) is to provide a lambda to [UseExceptionHandler](#). Using a lambda allows access to the error and writing a problem details response with [IProblemDetailsService.WriteAsync](#):

C#

```

using Microsoft.AspNetCore.Diagnostics;
using static System.Net.Mime.MediaTypeNames;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseExceptionHandler();
app.UseStatusCodePages();

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler(exceptionHandlerApp =>
    {

```

```

        exceptionHandlerApp.Run(async context =>
        {
            context.Response.StatusCode =
StatusCodes.Status500InternalServerError;
            context.Response.ContentType = Text.Plain;

            var title = "Bad Input";
            var detail = "Invalid input";
            var type = "https://errors.example.com/badInput";

            if (context.RequestServices.GetService<IProblemDetailsService>()
is
                { } problemDetailsService)
            {
                var exceptionHandlerFeature =
context.Features.Get<IExceptionHandlerFeature>();

                var exceptionType = exceptionHandlerFeature?.Error;
                if (exceptionType != null &&
                    exceptionType.Message.Contains("infinity"))
                {
                    title = "Argument exception";
                    detail = "Invalid input";
                    type = "https://errors.example.com/argumentException";
                }


                await problemDetailsService.WriteAsync(new
ProblemDetailsContext
                {
                    HttpContext = context,
                    ProblemDetails =
                    {
                        Title = title,
                        Detail = detail,
                        Type = type
                    }
                });
            }
        });
    });
}

app.MapControllers();
app.Run();

```

### Warning

Do **not** serve sensitive error information to clients. Serving errors is a security risk.

An alternative approach to generate problem details is to use the third-party NuGet package [Hellang.Middleware.ProblemDetails](#)  that can be used to map exceptions and