learn.microsoft.com

ASP.NET Core Middleware

Rick-Anderson

83–105 minutes

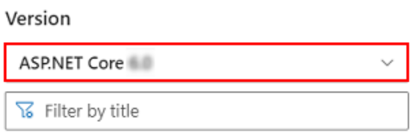
ASP.NET Core Middleware

- Article
- 05/03/2023
-

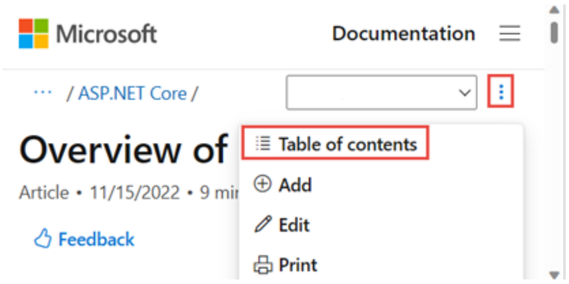
In this article

Note

This isn't the latest version of this article. To switch to the latest, use the ASP.NET Core version selector at the top of the table of contents.



If the selector isn't visible in a narrow browser window, widen the window or select the vertical ellipsis (⋮) > **Table of contents**.



By Rick Anderson and Steve Smith

Middleware is software that's assembled into an app pipeline to handle requests and responses. Each component:

- Chooses whether to pass the request to the next component in the pipeline.
- Can perform work before and after the next component in the pipeline.

Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

Request delegates are configured using Run, Map, and Use extension methods. An individual request delegate can be specified in-line as an anonymous method (called in-line middleware), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are *middleware*, also called *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the pipeline. When a middleware short-circuits, it's called a *terminal middleware* because it prevents further middleware from processing the request.

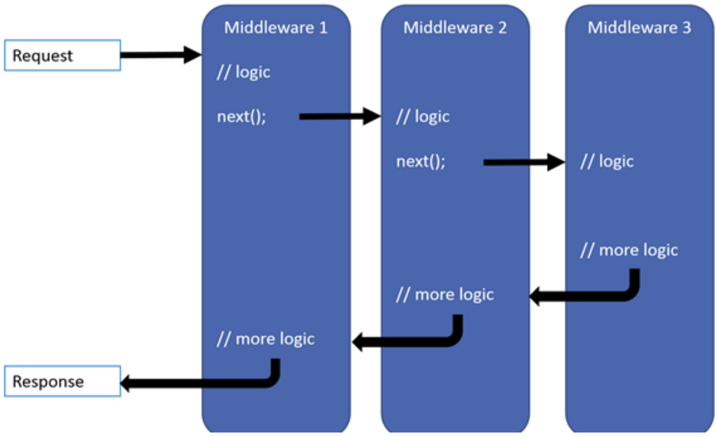
Migrate HTTP handlers and modules to ASP.NET Core middleware explains the difference between request pipelines in ASP.NET Core and ASP.NET 4.x and provides additional middleware samples.

Middleware code analysis

ASP.NET Core includes many compiler platform analyzers that inspect application code for quality. For more information, see Code analysis in ASP.NET Core apps

Create a middleware pipeline with `WebApplication`

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



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

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

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

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

app.Run();
```

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

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

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

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

app.Run();
```

Short-circuiting the request pipeline

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

Warning

Don't call `next.Invoke` after the response has been sent to the client. Changes to HttpResponse after the response has