# Understanding Middleware In ASP.NET Core

Anupam Maiti Oct 16, 2023 304.9k 8 20

## Introduction

This article demonstrates Middleware concepts in ASP.NET Core. At the end of this article, you will have a clear understanding of the below points.

- What is Middleware?
- Why is Middleware ordering important?
- Understanding of Run, Use, and Map Method.
- How to create a Custom Middleware?
- How to enable directory browsing through Middleware?

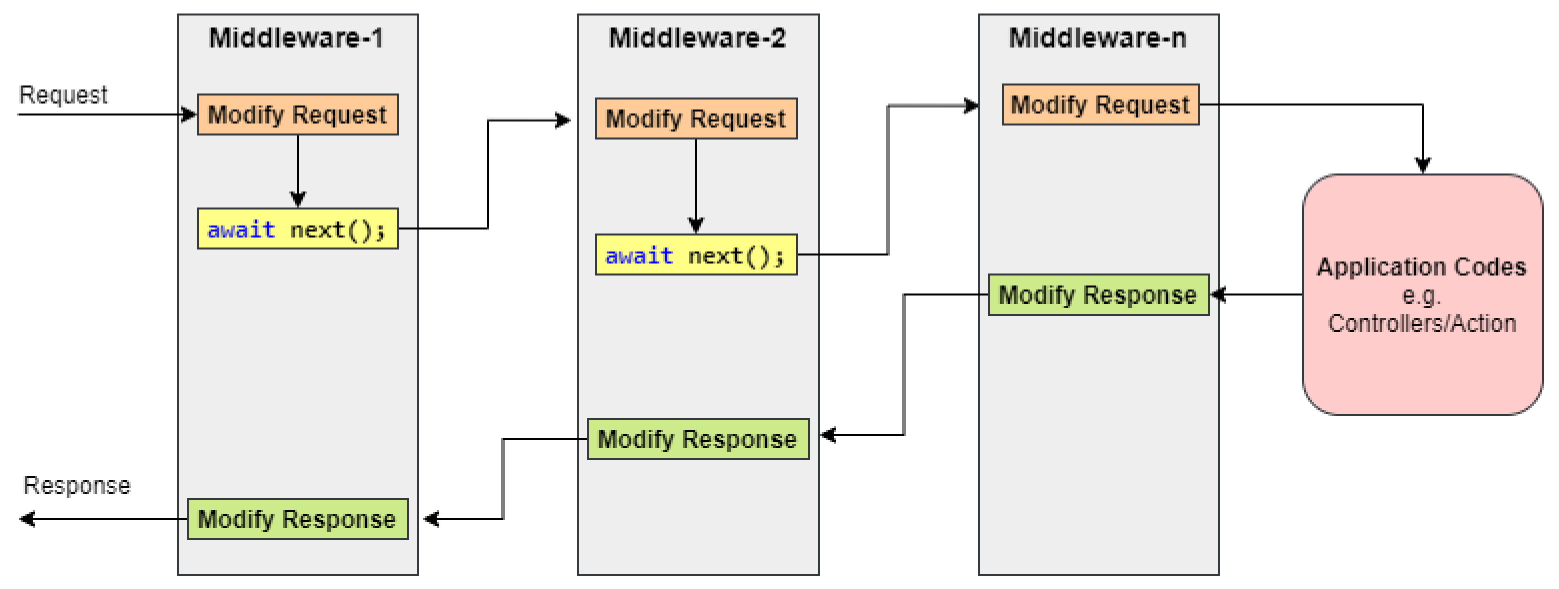
## What is Middleware?

Middleware is a piece of code in an application pipeline used to handle requests and responses.

For example, we may have a middleware component to authenticate a user, another piece of middleware to handle errors, and another middleware to serve static files such as JavaScript files, CSS files, images, etc.

Middleware can be built-in as part of the .NET Core framework, added via NuGet packages, or can be custom middleware. These middleware components are configured as part of the application startup class in the configure method. Configure methods set up a request processing pipeline for an ASP.NET Core application. It consists of a sequence of request delegates called one after the other.

The following figure illustrates how a request processes through middleware components.

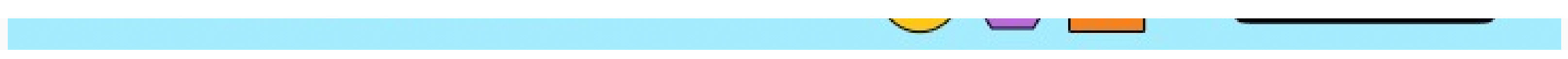


Generally, each middleware may handle the incoming requests and pass execution to the next middleware for further processing.

However, a middleware component can decide not to call the next piece of middleware in the pipeline. This is called short-circuiting or terminating the request pipeline. Short-circuiting is often desirable because it avoids unnecessary work. For example, if the request is for a static file like an image CSS file, JavaScript file, etc., these static files middleware can handle and serve that request and then short-circuit the rest of the pipeline.

Let’s create an ASP.NET Core Web application and observe the default configuration of middleware in the Configure method of the Startup class.

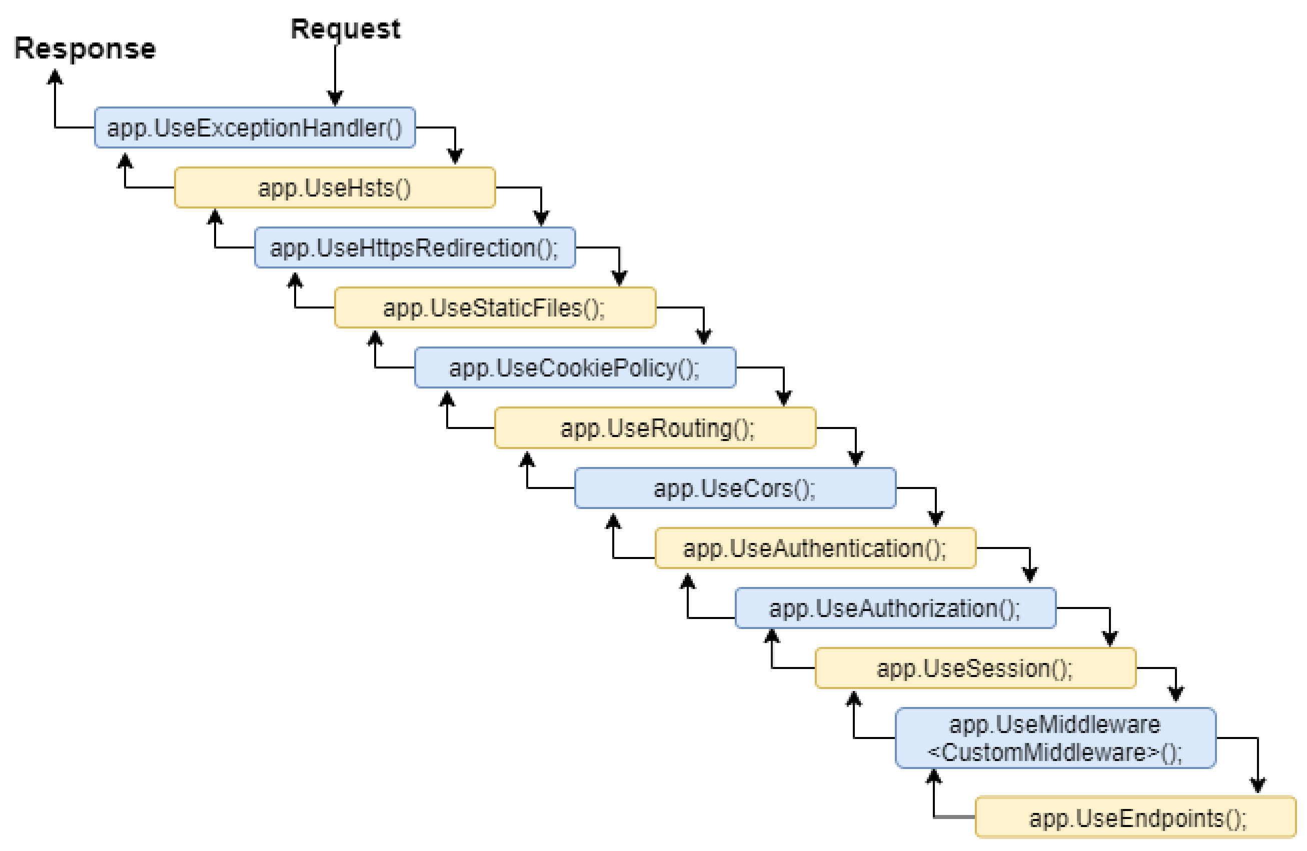
```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        // This middleware is used to report app runtime errors in a development envir
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // This middleware catches exceptions thrown in a production environment.
        app.UseExceptionHandler("/Error");
        // The default HSTS value is 30 days. You may want to change this for producti
        app.UseHsts(); // Adds the Strict-Transport-Security header.
    }

    // This middleware is used to redirects HTTP requests to HTTPS.
    app.UseHttpsRedirection();

    // This middleware is used to returns static files and short-circuits further requ
    app.UseStaticFiles();

    // This middleware is used to route requests.
    app.UseRouting();

    // This middleware is used to authorizes a user to access secure resources.
    app.UseAuthorization();

    // This middleware is used to add Razor Pages endpoints to the request pipeline.
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
```



## Middleware Ordering

Middleware components are executed in the order they are added to the pipeline, and care should be taken to add the middleware in the right order; otherwise, the application may not function as expected. This ordering is critical for security, performance, and functionality.

The following middleware components are for common app scenarios in the recommended order.



The first configured middleware has received the request, modified it (if required), and passed control to the next middleware. Similarly, the first middleware is executed at the last while processing a response if the echo comes back down the tube. That's why Exception-handling delegates need to be called early in the pipeline, so they can validate the result and display a possible exception in a browser and client-friendly way.

## Understanding the Run, Use, and Map Method

### app.Run() Method

This middleware component may expose Run[Middleware] methods that are executed at the end of the pipeline. Generally, this acts as a terminal middleware and is added at the end of the request pipeline, as it cannot call the next middleware.

### app.Use() Method

This is used to configure multiple middleware, unlike app.Run(), We can include the next parameter into it, which calls the next request delegate in the pipeline. We can also short-circuit (terminate) the pipeline by not calling the next parameter.

Let's consider the following example with the app.Use() and app.Run() and observe the output/response.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Before Invoke from 1st app.Use()\n");
        await next();
        await context.Response.WriteAsync("After Invoke from 1st app.Use()\n");
    });

    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Before Invoke from 2nd app.Use()\n");
        await next();
        await context.Response.WriteAsync("After Invoke from 2nd app.Use()\n");
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello from 1st app.Run()\n");
    });

    // the following will never be executed
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello from 2nd app.Run()\n");
    });
}
```

LocalhostView

The first app.Run() delegate terminates the pipeline. In the following example, only the first delegate ("Hello from 1st app.Run()") will run, and the request will never reach the second Run method.

### app.Map() Method

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

Let's consider the following example with the app.Map() and observe the output/response.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Map("/m1", HandleMapOne);
    app.Map("/m2", appMap => {
        appMap.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd app.Map()");
        });
    });
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello from app.Run()");
    });
}
private static void HandleMapOne(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Hello from 1st app.Map()");
    });
}
```

The following table shows the requests and responses from localhost using the above code.

| Request | Response |
|---|---|
| https://localhost:44362/ | Hello from app.Run() |
| https://localhost:44362/m1 | Hello from 1st app.Map() |
| https://localhost:44362/m1/xyz | Hello from 1st app.Map() |
| https://localhost:44362/m2 | Hello from 2nd app.Map() |
| https://localhost:44362/m500 | Hello from app.Run() |

## Creating a Custom Middleware

Middleware is generally encapsulated in a class and exposed with an extension method. The custom middleware can be built with a class with the InvokeAsync() method and RequestDelegate type parameter in the constructor. RequestDelegate type is required in order to execute the next middleware in a sequence.

Let's consider an example where we need to create custom middleware to log a request URL in a web application.

```
public class LogURLMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<LogURLMiddleware> _logger;
    public LogURLMiddleware(RequestDelegate next, ILoggerFactory loggerFactory)
    {
        _next = next;
        _logger = loggerFactory?.CreateLogger<LogURLMiddleware>() ??
        throw new ArgumentNullException(nameof(loggerFactory));
    }
    public async Task InvokeAsync(HttpContext context)
    {
        _logger.LogInformation($"Request URL: {Microsoft.AspNetCore.Http.Extensions.Ur
        await this._next(context);
    }
}
public static class LogURLMiddlewareExtensions
{
    public static IApplicationBuilder UseLogUrl(this IApplicationBuilder app)
    {
        return app.UseMiddleware<LogURLMiddleware>();
    }
}
```

In Configure method.

```
app.UseLogUrl();
```

## Enabling directory browsing through Middleware

Directory browsing allows users of your web app to see a directory listing and files within a specified directory.

Directory browsing is disabled by default for security reasons.

Let's consider an example where we want to allow the list of images in the browser under the images folder in wwwroot. UseDirectoryBrowser middleware can handle and serve those images for that request and then short-circuit the rest of the pipeline.

```
app.UseDirectoryBrowser(new DirectoryBrowserOptions
{
    FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory
    RequestPath = "/images"
});
```

Directory

## Summary

So, Middleware in ASP.NET Core controls how our application responds to HTTP requests.

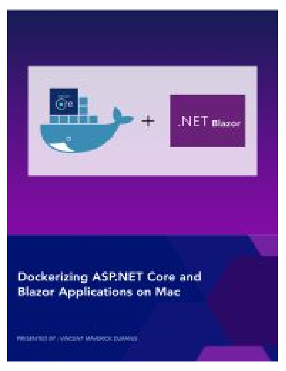
In summary, every middleware component in ASP.NET Core.

- Has access to both the incoming requests and the outgoing response.
- May simply pass the request to the next piece of middleware in the pipeline.
- May perform some processing logic and then pass that request to the next middleware for further processing.
- May terminate(short-circuit) the request pipeline whenever required.
- They are executed in the order they are added to the pipeline.

I hope you gained some insights into this topic! Happy Learning!

ASP.NET Core Custom Middleware Middleware in .NET Core Middleware In ASP.NET Core Middleware Ordering

RECOMMENDED FREE EBOOK



Dockerizing ASP.NET Core and Blazor Applications on Mac

Download Now!

SIMILAR ARTICLES

Error Management in .NET Core
ASP.NET Core - Custom Middleware
Learn About Middleware In ASP.NET Core
Passing Parameters To Middleware In ASP.NET Core 2.0
Create a Custom Middleware Component in ASP.NET Core

Anupam Maiti TOP 500 177 3.4m 4

Anupam is a technology enthusiast with a positive attitude, constantly seeking growth and learning opportunities. His expertise lies in successfully executing challenging Cloud Assessment & Migration, Cloud Native En... Read more

8 View All Comments

Type your comment here and press Enter Key (Minimum 10 characters)