ASP.NET Core Middleware

07/15/2020 • 22 minutes to read •  +6

In this article

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.

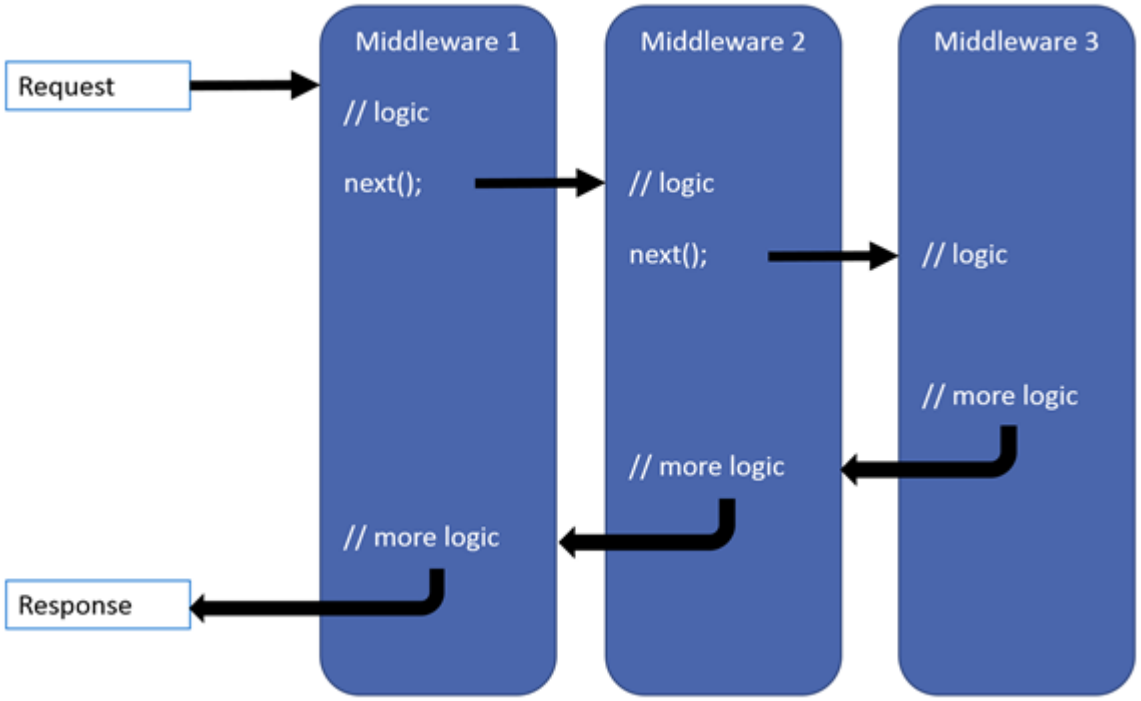
Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

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.

Create a middleware pipeline with IApplicationBuilder

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



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

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

C# Copy

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello, World!");
        });
    }
}
```

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

C# Copy

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't 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.");
        });
    }
}
```

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 started throw an exception. For example, **setting headers and a status code throw an exception**. Writing to the response body after calling `next`:
>
> - May cause a protocol violation. For example, writing more than the stated `Content-Length`.
> - May corrupt the body format. For example, writing an HTML footer to a CSS file.
>
> **HasStarted** is a useful hint to indicate if headers have been sent or the body has been written to.

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

C# Copy

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

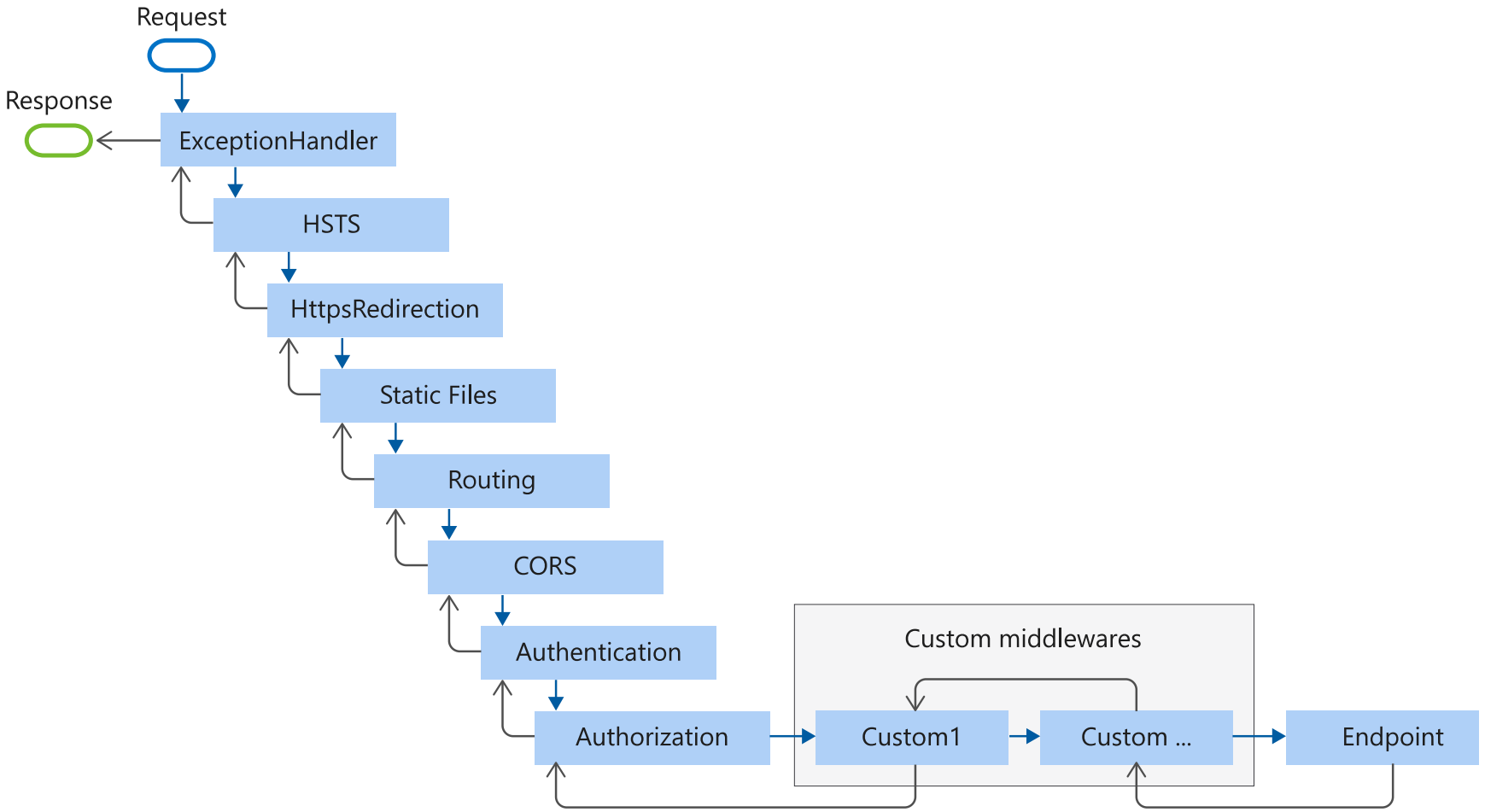
```
            app.Run(async context =>
            {
                await context.Response.WriteAsync("Hello from 2nd delegate.");
            });
        }
    }
```

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

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

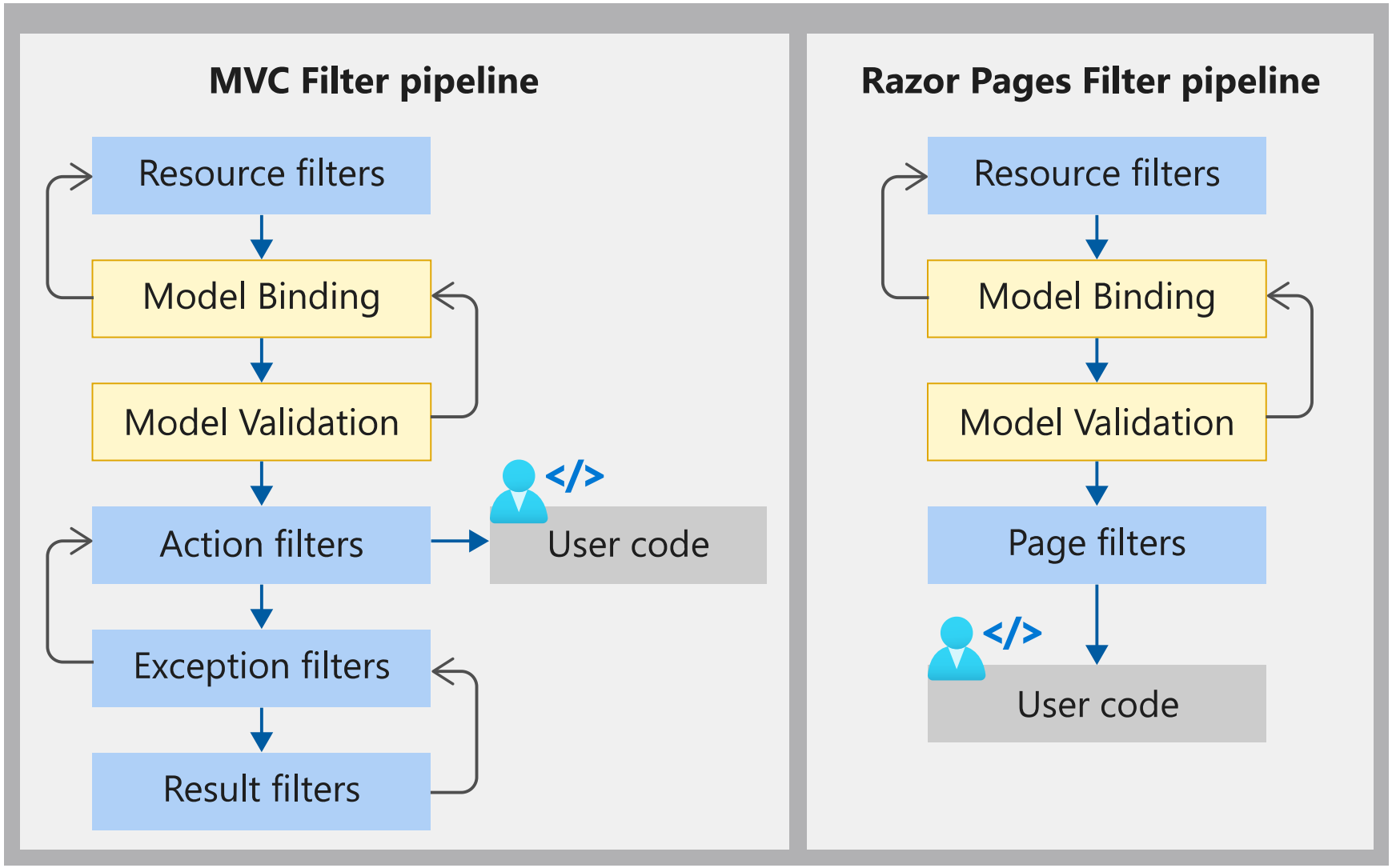
Middleware order

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



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

MVC Endpoint
(called by the Endpoint Middleware)



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

The following `Startup.Configure` method adds security-related middleware components in the recommended order:

C# Copy

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

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

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

    app.UseAuthentication();
    app.UseAuthorization();
    // app.UseSession();
    // app.UseResponseCaching();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

In the preceding code:

- Middleware that is not added when creating a new web app with individual users accounts is commented out.
- Not every middleware needs to go in this exact order, but many do. For example:
  - `UseCors`, `UseAuthentication`, and `UseAuthorization` must go in the order shown.
  - `UseCors` currently must go before `UseResponseCaching` due to this bug.

The following `Startup.Configure` method adds middleware components for common app scenarios:

1. Exception/error handling
   - When the app runs in the Development environment:
     - Developer Exception Page Middleware (UseDeveloperExceptionPage) reports app runtime errors.
     - Database Error Page Middleware reports database runtime errors.
   - When the app runs in the Production environment:
     - Exception Handler Middleware (UseExceptionHandler) catches exceptions thrown in the following middlewares.
     - HTTP Strict Transport Security Protocol (HSTS) Middleware (UseHsts) adds the `Strict-Transport-Security` header.
2. HTTPS Redirection Middleware (UseHttpsRedirection) redirects HTTP requests to HTTPS.
3. Static File Middleware (UseStaticFiles) returns static files and short-circuits further request processing.
4. Cookie Policy Middleware (UseCookiePolicy) conforms the app to the EU General Data Protection Regulation (GDPR) regulations.
5. Routing Middleware (UseRouting) to route requests.
6. Authentication Middleware (UseAuthentication) attempts to authenticate the user before they're allowed access to secure resources.
7. Authorization Middleware (UseAuthorization) authorizes a user to access secure resources.
8. Session Middleware (UseSession) establishes and maintains session state. If the app uses session state, call Session Middleware after Cookie Policy Middleware and before MVC Middleware.

9. Endpoint Routing Middleware (UseEndpoints with MapRazorPages) to add Razor Pages endpoints to the request pipeline.

C# Copy

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

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();
    app.UseSession();

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

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

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

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

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

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

C# Copy

```
public void Configure(IApplicationBuilder app)
{
    // Static files aren't compressed by Static File Middleware.
    app.UseStaticFiles();

    app.UseResponseCompression();

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

For Single Page Applications (SPAs), the SPA middleware UseSpaStaticFiles usually comes last in the middleware pipeline. The SPA middleware comes last:

- To allow all other middlewares to respond to matching requests first.
- To allow SPAs with client-side routing to run for all routes that are unrecognized by the server app.

For more details on SPAs, see the guides for the React and Angular project templates.

Forwarded Headers Middleware order

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

Branch the middleware pipeline

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

C# Copy

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

    private static void HandleMapTest2(IApplicationBuilder app)
```

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

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1", HandleMapTest1);

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

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

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

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

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

`Map` supports nesting, for example:

C# Copy

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

`Map` can also match multiple segments at once:

C# Copy

```
public class Startup
{
    private static void HandleMultiSeg(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map multiple segments.");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1/seg1", HandleMultiSeg);

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

```
    }
}
```

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

C# Copy

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

    public void Configure(IApplicationBuilder app)
    {
        app.MapWhen(context => context.Request.Query.ContainsKey("branch"),
                               HandleBranch);

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

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

Request	Response

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

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

C# Copy

```
public class Startup
{
    private void HandleBranchAndRejoin(IApplicationBuilder app, ILogger<Startup> logger)
    {
        app.Use(async (context, next) =>
        {
            var branchVer = context.Request.Query["branch"];
            logger.LogInformation("Branch used = {branchVer}", branchVer);

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

    public void Configure(IApplicationBuilder app, ILogger<Startup> logger)
    {
        app.UseWhen(context => context.Request.Query.ContainsKey("branch"),
                               appBuilder => HandleBranchAndRejoin(appBuilder, logger));

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from main pipeline.");
```

```
            });
        }
    }
```

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

Built-in middleware

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

Middleware	Description	Order
Authentication	Provides authentication support.	Before `HttpContext.User` is needed. Terminal for OAuth callbacks.
Authorization	Provides authorization support.	Immediately after the Authentication Middleware.
Cookie Policy	Tracks consent from users for storing personal information and enforces minimum standards for cookie fields, such as `secure` and `SameSite`.	Before middleware that issues cookies. Examples: Authentication, Session, MVC (TempData).
CORS	Configures Cross-Origin Resource Sharing.	Before components that use CORS. `UseCors` currently must go before `UseResponseCaching` due to this bug.

Middleware	Description	Order
Diagnostics	Several separate middlewares that provide a developer exception page, exception handling, status code pages, and the default web page for new apps.	Before components that generate errors. Terminal for exceptions or serving the default web page for new apps.
Forwarded Headers	Forwards proxied headers onto the current request.	Before components that consume the updated fields. Examples: scheme, host, client IP, method.
Health Check	Checks the health of an ASP.NET Core app and its dependencies, such as checking database availability.	Terminal if a request matches a health check endpoint.
Header Propagation	Propagates HTTP headers from the incoming request to the outgoing HTTP Client requests.	
HTTP Method Override	Allows an incoming POST request to override the method.	Before components that consume the updated method.
HTTPS Redirection	Redirect all HTTP requests to HTTPS.	Before components that consume the URL.
HTTP Strict Transport Security (HSTS)	Security enhancement middleware that adds a special response header.	Before responses are sent and after components that modify requests. Examples: Forwarded Headers, URL Rewriting.
MVC	Processes requests with MVC/Razor Pages.	Terminal if a request matches a route.
OWIN	Interop with OWIN-based apps, servers, and middleware.	Terminal if the OWIN Middleware fully processes the request.

Middleware	Description	Order
Response Caching	Provides support for caching responses.	Before components that require caching. `UseCORS` must come before `UseResponseCaching`.
Response Compression	Provides support for compressing responses.	Before components that require compression.
Request Localization	Provides localization support.	Before localization sensitive components.
Endpoint Routing	Defines and constrains request routes.	Terminal for matching routes.
SPA	Handles all requests from this point in the middleware chain by returning the default page for the Single Page Application (SPA)	Late in the chain, so that other middleware for serving static files, MVC actions, etc., takes precedence.
Session	Provides support for managing user sessions.	Before components that require Session.
Static Files	Provides support for serving static files and directory browsing.	Terminal if a request matches a file.
URL Rewrite	Provides support for rewriting URLs and redirecting requests.	Before components that consume the URL.
WebSockets	Enables the WebSockets protocol.	Before components that are required to accept WebSocket requests.

Additional resources

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

---

Is this page helpful?

Yes No
