**Middleware in ASP .NET Core 3.1**

By Shahed C on March 30, 2020

[4 Replies](https://wakeupandcode.com/middleware-in-asp-net-core-3-1/#comments)



This is the thirteenth of a new [series of posts](https://wakeupandcode.com/aspnetcore/#aspnetcore2020) on ASP .NET Core 3.1 for 2020. In this series, we’ll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the [2019 series](https://wakeupandcode.com/aspnetcore/#aspnetcore2019), the 2020 series will mostly focus on a growing single codebase ([NetLearner!](https://wakeupandcode.com/netlearner-on-asp-net-core-3-1/)) instead of new unrelated code snippets week.

Previous post:

* [Logging in ASP .NET Core 3.1](https://wakeupandcode.com/logging-in-asp-net-core-3-1/)

**NetLearner on GitHub**:

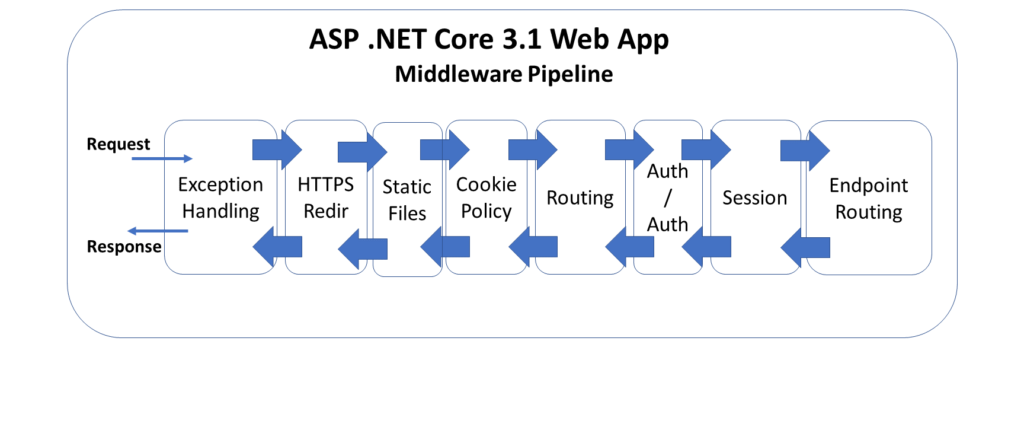
* Repository: <https://github.com/shahedc/NetLearnerApp>
* v0.13-alpha release: <https://github.com/shahedc/NetLearnerApp/releases/tag/v0.13-alpha>

**In this Article:**

* M is for Middleware in ASP .NET Core
* How It Works
* Built-in Middleware
* Branching out with Run, Map & Use
* References

**M is for Middleware in ASP .NET Core**

If you’ve been following my blog series (or if you’ve done any work with ASP .NET Core at all), you’ve already worked with the Middleware pipeline. When you create a new project using one of the built-in templates, your project is already supplied with a few calls to add/configure middleware services and then use them. This is accomplished by adding the calls to the Startup.cs configuration methods.

ASP .NET Core 3.1 Middleware Pipeline

The above diagram illustrates the typical order of middleware layers in an ASP .NET Core web application. The order is very important, so it is necessary to understand the placement of each *request delegate* in the pipeline.

* Exception Handling
* HTTPS Redirection
* Static Files
* Cookie Policy
* Routing
* Authentication
* Authorization
* Session
* Endpoint Routing

**How It Works**

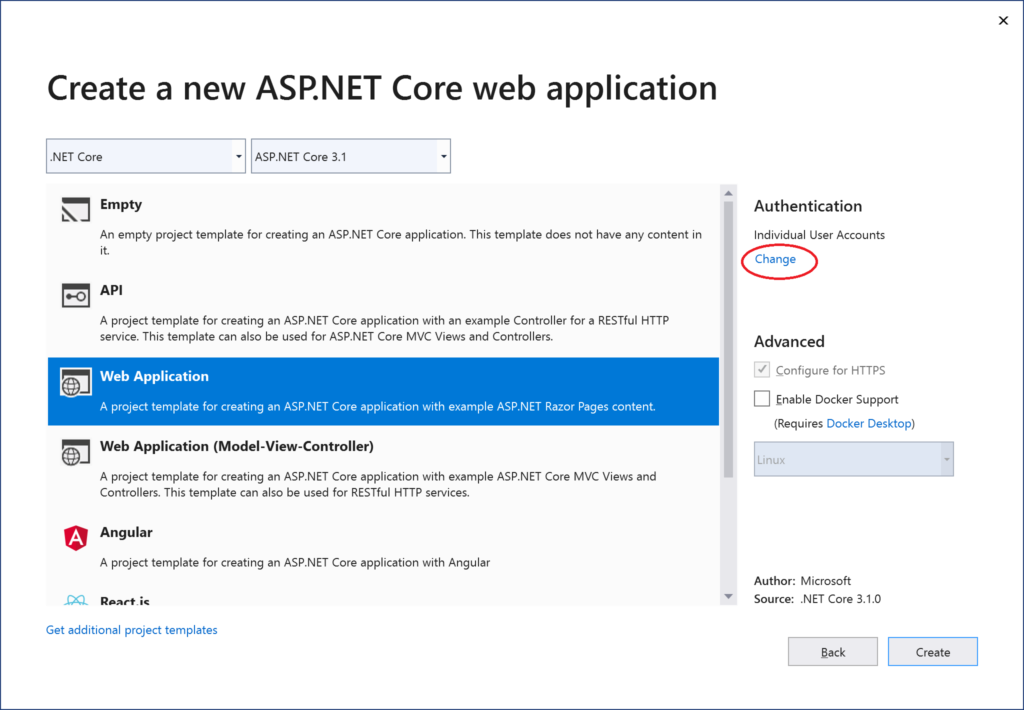
When an HTTP request comes in, the first request delegate handles that request. It can either pass the request down to the next in line or short-circuit the pipeline by preventing the request from propagating further. This is use very useful across multiple scenarios, e.g. serving static files without the need for authentication, handling exceptions before anything else, etc.

The returned response travels back in the reverse direction back through the pipeline. This allows each component to run code both times: when the request arrives and also when the response is on its way out.

Here’s what the **Configure**() method may look like, in a template-generated [Startup.cs](https://github.com/shahedc/NetLearnerApp/blob/master/src/NetLearner.Portal/Startup.cs) file:

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  
{  
 if (env.IsDevelopment())  
 {  
 app.UseDeveloperExceptionPage();  
 }  
 else  
 {  
 app.UseExceptionHandler("/Error");  
 app.UseHsts();  
 }  
  
 app.UseHttpsRedirection();  
 app.UseStaticFiles();  
 app.UseCookiePolicy(); // manually added  
  
 app.UseAuthentication();  
 app.UseAuthorization();  
  
 app.UseSession(); // manually added  
  
 app.UseEndpoints(endpoints =>  
 {  
 // map MVC routes, Razor Pages, Blazor Hub  
 });  
}

You may add calls to **UseCookiePolicy()** and **UseSession**()  manually, as these aren’t included in the template-generated project. The endpoint configuration will vary depending on the type of project template you start with: [MVC](https://github.com/shahedc/NetLearnerApp/blob/master/src/NetLearner.Mvc/Startup.cs), [Razor Pages](https://github.com/shahedc/NetLearnerApp/blob/master/src/NetLearner.Pages/Startup.cs) or [Blazor](https://github.com/shahedc/NetLearnerApp/blob/master/src/NetLearner.Blazor/Startup.cs), which you can mix and match. You may also enable authentication and HTTPS when creating the template.

 Adding a new project in VS2019

In order to configure the use of the Session middleware, you may add the following code in your **ConfigureServices**() method:

public void ConfigureServices(IServiceCollection services)  
{  
...  
 services.AddDistributedMemoryCache();  
 services.AddSession(options =>  
 {  
 options.IdleTimeout = TimeSpan.FromSeconds(10);  
 options.Cookie.HttpOnly = true;  
 options.Cookie.IsEssential = true;  
 });  
...  
}

The calls to **AddDistributedMemoryCache**() and **AddSession**() ensure that we have enabled a (memory cache) backing store for the session and then prepared the Session middleware for use.

**Built-In Middleware**

The information below explains how the built-in middleware works, and why the order is important. The ***UseXYZ***() methods are merely extension methods that are prefixed with the word “Use” as a useful convention, making it easy to discover Middleware components when typing code. Keep this in mind mind when developing custom middleware.

***Exception Handling*:**

* **UseDeveloperExceptionPage**() & **UseDatabaseErrorPage**(): used in *development* to catch run-time exceptions
* **UseExceptionHandler**(): used in *production* for run-time exceptions

Calling these methods first ensures that exceptions are caught in any of the middleware components that follow. For more information, check out the detailed post on [Handling Errors in ASP .NET Core](https://wakeupandcode.com/handling-errors-in-asp-net-core-3-1/), earlier in this series.

***HSTS & HTTPS Redirection*:**

* **UseHsts**(): used in production to enable HSTS (HTTP Strict Transport Security Protocol) and enforce HTTPS.
* **UseHttpsRedirection**(): forces HTTP calls to automatically redirect to equivalent HTTPS addresses.

Calling these methods next ensure that HTTPS can be enforced before resources are served from a web browser. For more information, check out the detailed post on [Protocols in ASP .NET Core: HTTPS and HTTP/2](https://wakeupandcode.com/protocols-in-asp-net-core-https-and-http2/).

***Static Files:***

* **UseStaticFiles**(): used to enable static files, such as HTML, JavaScript, CSS and graphics files. Called early on to avoid the need for authentication, session or MVC middleware.

Calling this before authentication ensures that static files can be served quickly without unnecessarily triggering authentication middleware. For more information, check out the detailed post on [JavaScript, CSS, HTML & Other Static Files in ASP .NET Core](https://wakeupandcode.com/javascript-css-html-other-static-files-in-asp-net-core-3-1/).

***Cookie Policy:***

* **UseCookiePolicy**(): used to enforce cookie policy and display GDPR-friendly messaging

Calling this before the next set of middleware ensures that the calls that follow can make use of cookies if consented. For more information, check out the detailed post on [Cookies and Consent in ASP .NET Core](https://wakeupandcode.com/cookies-and-consent-in-asp-net-core-3-1/).

***Authentication, Authorization & Sessions***:

* **UseAuthentication**(): used to enable authentication and then subsequently allow authorization.
* **UseSession**(): manually added to the Startup file to enable the Session middleware.

Calling these after cookie authentication (but before the endpoint routing middleware) ensures that cookies can be issued as necessary and that the user can be authenticated before the endpoint routing kicks in. For more information, check out the detailed post on [Authentication & Authorization in ASP .NET Core](https://wakeupandcode.com/authentication-authorization-in-asp-net-core-3-1/).

***Endpoint Routing:***

* **UseEndpoints**(): usage varies based on project templates used for MVC, Razor Pages and Blazor.
* endpoints.**MapControllerRoute**(): set the default route and any custom routes when using MVC.
* endpoints.**MapRazorPages**(): sets up default Razor Pages routing behavior
* endpoints.**BlazorHub**(): sets up Blazor Hub

To create your own *custom* middleware, check out the official docs at:

* Write custom ASP.NET Core middleware: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/write>

**Branching out with Run, Map & Use**

The so-called “request delegates” are made possible with the implementation of three types of extension methods:

* [Run](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.runextensions.run?view=aspnetcore-3.1): enables short-circuiting with a terminal middleware delegate
* [Map](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.mapextensions.map?view=aspnetcore-3.1): allows branching of the pipeline path
* [Use](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.useextensions.use): call the next desired middleware delegate

If you’ve tried out the absolute basic example of an ASP .NET Core application (generated from the Empty template), you may have seen the following syntax in your [Startup.cs](https://github.com/shahedc/NetLearnerApp/blob/master/src/NetLearner.EmptySample/Startup.cs) file.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  
{  
 ...  
 app.UseEndpoints(endpoints =>   
 {  
 endpoints.MapGet("/", async (context) =>  
 {  
 await context.Response.WriteAsync("Hello World!");  
 });  
 });   
}

You can use an alternative approach to terminate the pipeline, by calling **app.Use**() and then triggering the next middleware component with a call to **next(context)** as shown below. If there is no call to **next**(), then it essentially short-circuits the middleware pipeline.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  
{  
 // ...  
 // Approach 1: terminal middleware  
 app.Use(next => async context =>  
 {  
 if (context.Request.Path == "/")  
 {  
 await context.Response.WriteAsync("Hello and goodbye!");  
 return;  
 }  
 await next(context);  
 });  
 app.UseRouting();  
 app.UseEndpoints(...); // Approach 2: routing  
}

Finally, there is the **Map**() method, which creates separate forked paths/branches for your middleware pipeline and multiple terminating ends. Check out the official documentation on mapped branches for more information:

* Branch middleware pipeline: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1#branch-the-middleware-pipeline>

In the **Configure**() method, each call to **app.Map**() establishes a separate branch that can be triggered with the appropriate relative path. Each handler method then contains its own terminating map.Run() method.

In the official sample:

* accessing /*map1* in an HTTP request will call **HandleMapTest1**()
* accessing /*map2* in an HTTP request will call **HandleMapTest2**()
* accessing the / root of the web application will call the default terminating **Run**() method
* specifying an invalid path will also call the default **Run**() method

**References**

* ASP.NET Core Middleware: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware>
* Write custom ASP.NET Core middleware: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/write>
* Session and app state in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/app-state>