**Middleware in ASP .NET Core**

By Shahed C on April 3, 2019

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

This is the **thirteenth**of a [series of posts](https://wakeupandcode.com/aspnetcore/#aspnetcore2019) on ASP .NET Core in 2019. In this series, we’ll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**

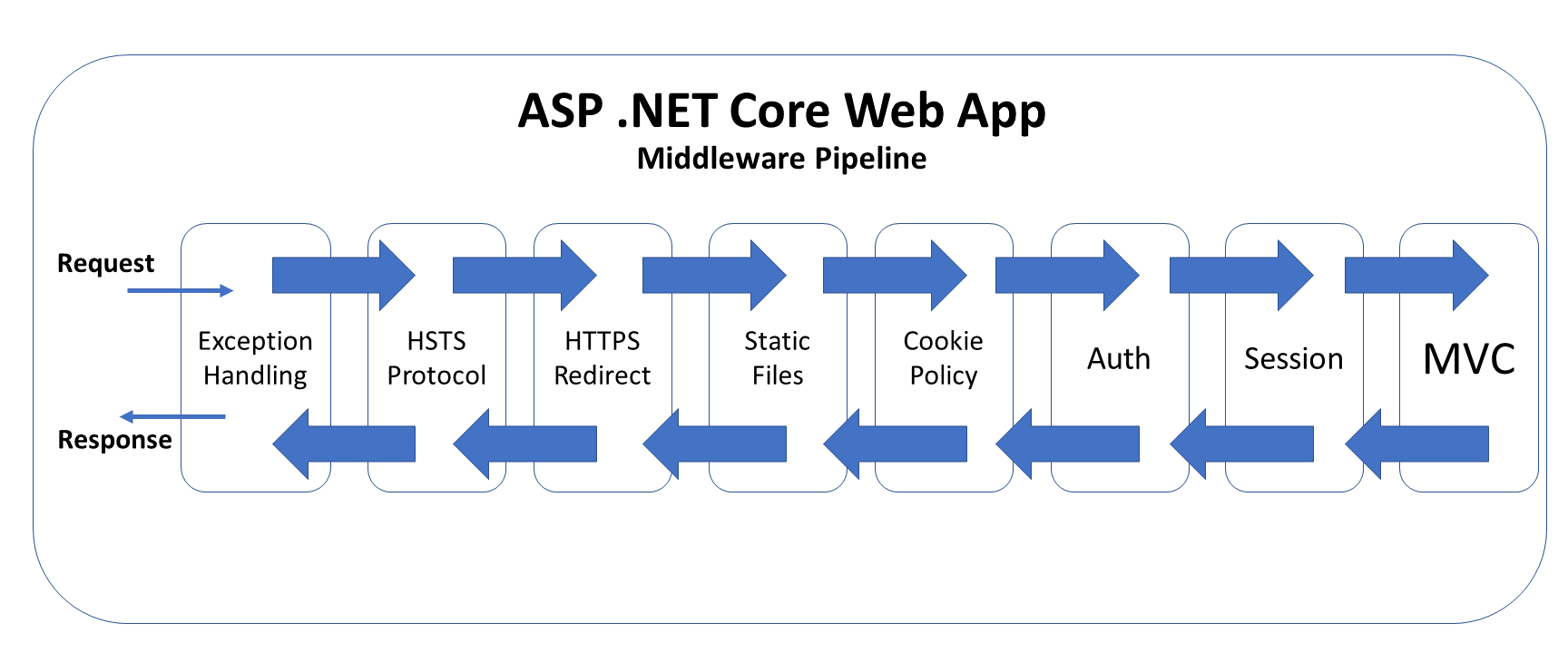
** A – Z of ASP .NET Core!**

**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.

[](https://wakeupandcode.com/wp-content/uploads/2019/04/Blog-Diagram-Middleware-Pipeline.png)

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.

To follow along, take a look at the sample project on Github:

 Middleware Sample: <https://github.com/shahedc/AspNetCoreMiddlewareSample>

**How It Works**

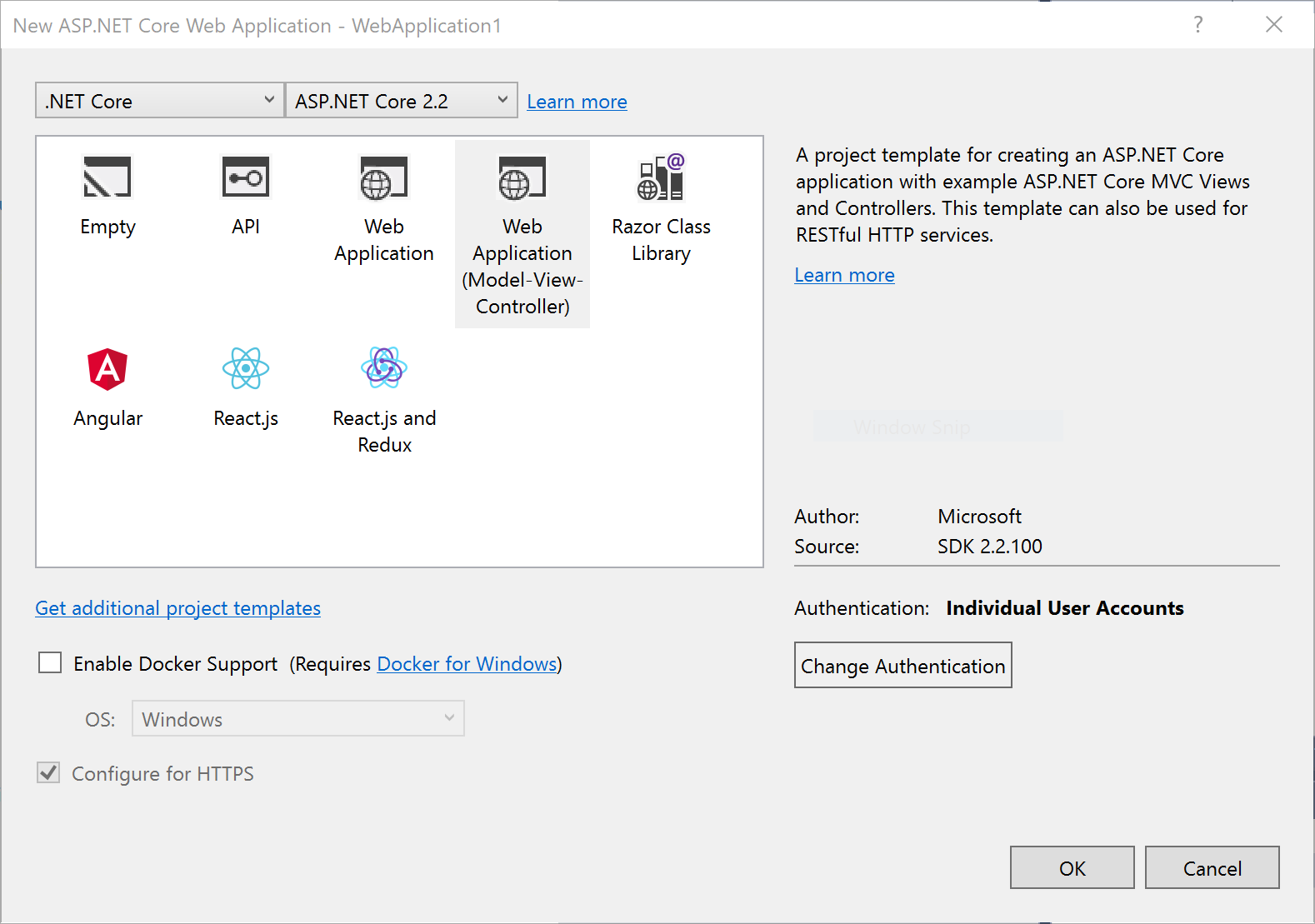
When an HTTP request comes in, the first request delegate handles that request. It can either pass the request down to the next in line or short-circuit the pipeline by preventing the request from propagating further. This is use very useful across multiple scenarios, e.g. serving static files without 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 looks like, in a template-generated [Startup.cs](https://github.com/shahedc/AspNetCoreMiddlewareSample/blob/master/MiddlewareSampleMvc/Startup.cs) file:

public void Configure(IApplicationBuilder app, IHostingEnvironment env)  
{  
 if (env.IsDevelopment())  
 {  
 app.UseDeveloperExceptionPage();  
 app.UseDatabaseErrorPage();  
 }  
 else  
 {  
 app.UseExceptionHandler("/Home/Error");  
 app.UseHsts();  
 }  
  
 app.UseHttpsRedirection();  
 app.UseStaticFiles();  
 app.UseCookiePolicy();  
  
 app.UseAuthentication();  
  
 app.UseSession();  
  
 app.UseMvc(routes =>  
 {  
 routes.MapRoute(  
 name: "default",  
 template: "{controller=Home}/{action=Index}/{id?}");  
 });  
}

I have added a call to **UseSession**()  to call the Session Middleware, as it wasn’t included in the template-generated project. I also enabled authentication and HTTPS when creating the template.

[](https://wakeupandcode.com/wp-content/uploads/2019/04/VS2017-Project-New.png)

In order to configure the use of the Session middleware, I also had to add the following code in my **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;  
 });  
  
 services.AddMvc();  
  
...  
  
}

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. In the [Razor Pages version of Startup.cs](https://github.com/shahedc/AspNetCoreMiddlewareSample/blob/master/MiddlewareSampleRazor/Startup.cs), the **Configure**() method is a little simpler, as it doesn’t need to set up MVC routes for controllers in the call to **useMvc**().

**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/), 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-static-files-in-asp-net-core/).

***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/).

***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 MVC middleware) ensures that cookies can be issued as necessary and that the user can be authenticated before the MVC engine 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-razor-pages/).

***MVC & Routing:***

* **UseMvc**(): enables the use of MVC in your web application, with the ability to customize routes for your MVC application and set other options.
* routes.**MapRoute**(): set the default route and any custom routes when using MVC.

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-2.2): enables short-circuiting with a terminal middleware delegate
* [Map](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.mapextensions.map): 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/AspNetCoreMiddlewareSample/blob/master/EmptyWeb/Startup.cs) file. Here, the **app.Run**() method terminates the pipeline, so you wouldn’t add any additional middleware calls after it.

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

The call to **app.Use**() can be used to trigger the next middleware component with a call to **next.Invoke**() as shown below:

public void Configure(IApplicationBuilder app)  
{  
 app.Use(async (context, next) =>  
 {  
 // ...  
 await next.Invoke();  
 // ...  
 });  
  
 app.Run(...);  
}

If there is no call to **next.Invoke**(), then it essentially short-circuits the middleware pipeline. Finally, there is the **Map**() method, which creates separate forked paths/branches for your middleware pipeline and multiple terminating ends. The code snippet below shows the contents of a [Startup.cs](https://github.com/shahedc/AspNetCoreMiddlewareSample/blob/master/MappedBranches/Startup.cs) file with mapped branches:

// first branch handler  
private static void HandleBranch1(IApplicationBuilder app)  
{  
 app.Run(async context =>  
 {  
 await context.Response.WriteAsync("Branch 1.");  
 });  
}  
  
// second branch handler  
private static void HandleBranch2(IApplicationBuilder app)  
{  
 app.Run(async context =>  
 {  
 await context.Response.WriteAsync("Branch 2.");  
 });  
}  
  
public void Configure(IApplicationBuilder app)  
{  
 app.Map("/mappedBranch1", HandleBranch1);  
  
 app.Map("/mappedBranch2", HandleBranch2);  
  
 // default terminating Run() method  
 app.Run(async context =>  
 {  
 await context.Response.WriteAsync("Terminating Run() method.");  
 });  
}

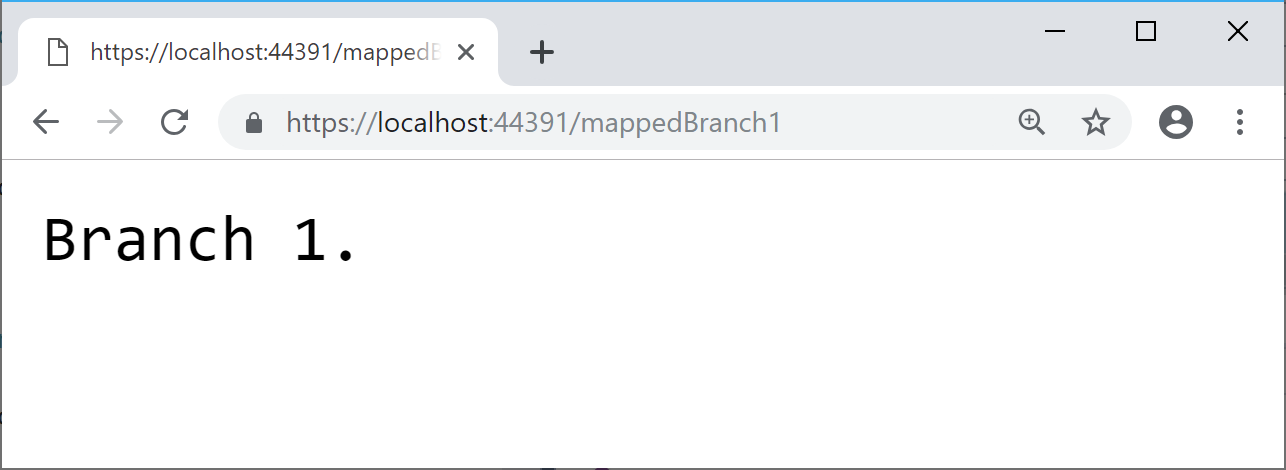
In the **Configure**() method, each call to **app.Map**() establishes a separate branch that can be triggered with the appropriate relative path. Each handler method then contains its own terminating map.Run() method.

For example:

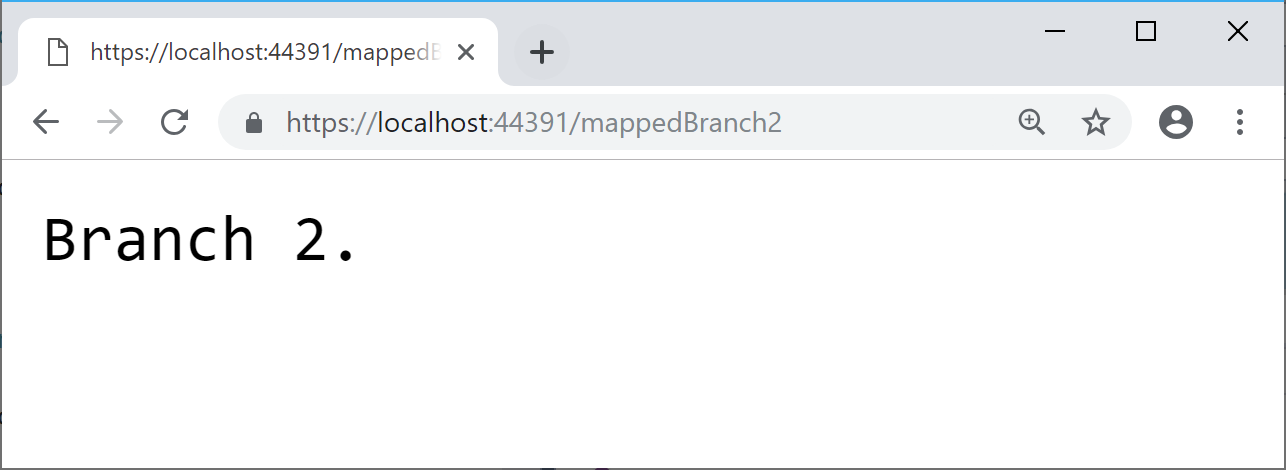
* accessing /*mappedBranch1* in an HTTP request will call **HandleBranch1**()
* accessing /*mappedBranch2* in an HTTP request will call **HandleBranch2**()
* accessing the / root of the web application will call the default terminating **Run**() method
* specifying an invalid path will also call the default **Run**() method

The screenshots below illustrate the results of the above scenarios:

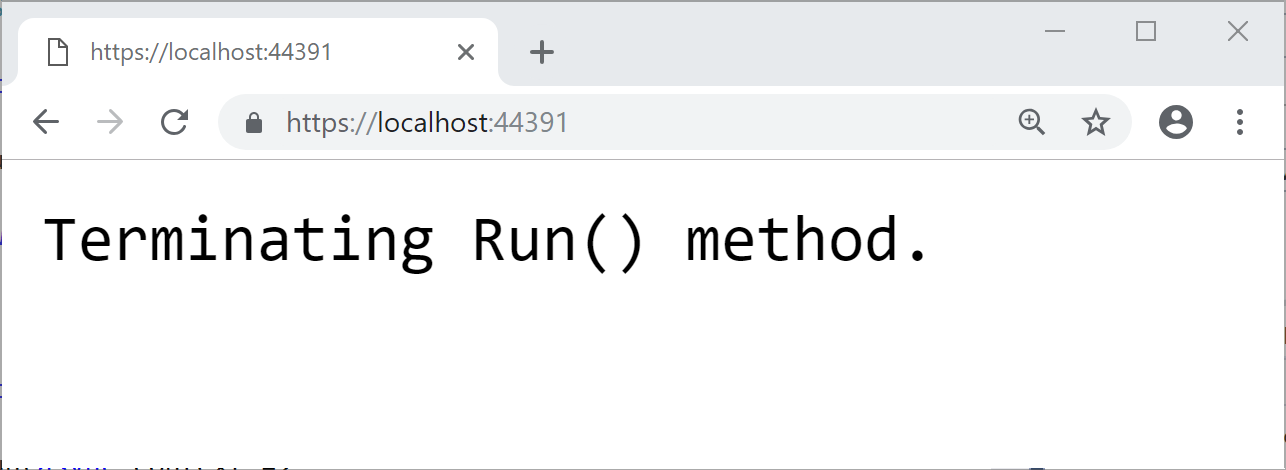
*Scenario 1*: https://localhost:44391/mappedBranch1

[](https://wakeupandcode.com/wp-content/uploads/2019/04/Branch-1.png)

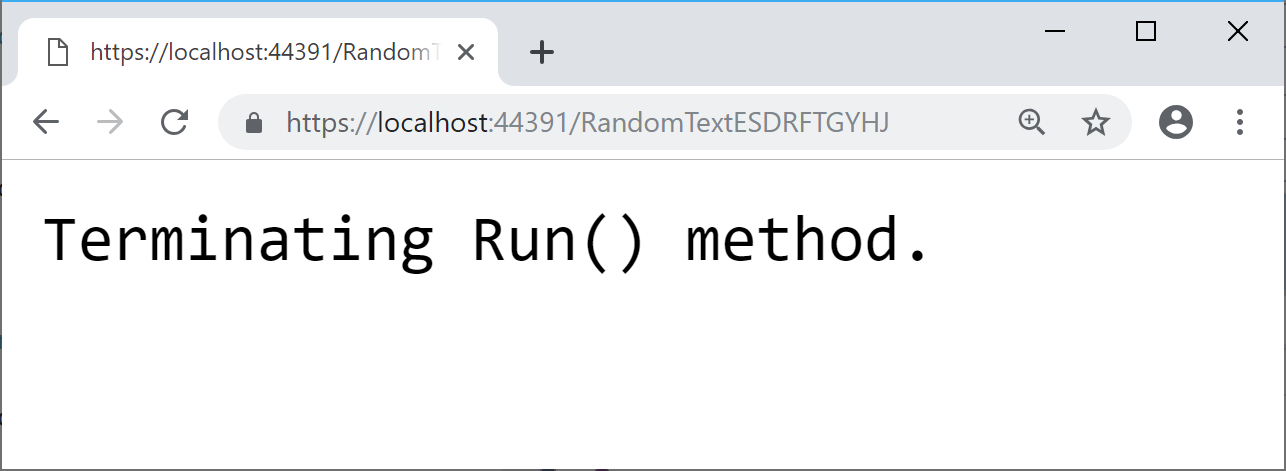
*Scenario 2*: https://localhost:44391/mappedBranch2

[](https://wakeupandcode.com/wp-content/uploads/2019/04/Branch-2.png)

*Scenario 3*: https://localhost:44391/

[](https://wakeupandcode.com/wp-content/uploads/2019/04/Branch-Root.png)

*Scenario 4*: https://localhost:44391/RandomTextESDRFTGYHJ

[](https://wakeupandcode.com/wp-content/uploads/2019/04/Branch-Random.png)

**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>

Hope you enjoyed learning about Middleware in ASP.NET Core. Here is some feedback I received via Twitter: *“Learned so [much] about #ASPNetCore  middleware from this article. The templates should have comments in them with this stuff!”*

* Tweet: <https://twitter.com/SturlaThorvalds/status/1114802739744321536>

Learned so mich about [#ASPNetCore](https://twitter.com/hashtag/ASPNetCore?src=hash&amp;ref_src=twsrc%5Etfw) middleware fro. This article. The templates should have comments in them with this stuff! <https://t.co/0au87iDjHl>

— Sturla Þorvaldsson (@SturlaThorvalds) [April 7, 2019](https://twitter.com/SturlaThorvalds/status/1114802739744321536?ref_src=twsrc%5Etfw)

This entry was posted in [ASP.NET](https://wakeupandcode.com/category/asp-net/), [Learn](https://wakeupandcode.com/category/learn/), [Visual Studio](https://wakeupandcode.com/category/visual-studio/), [Web Development](https://wakeupandcode.com/category/web-development/) and tagged [.net](https://wakeupandcode.com/tag/net/), [.NET Core](https://wakeupandcode.com/tag/net-core/), [App Service](https://wakeupandcode.com/tag/app-service/), [ASP.NET](https://wakeupandcode.com/tag/asp-net/), [ASP.NET Core](https://wakeupandcode.com/tag/asp-net-core/), [Middleware](https://wakeupandcode.com/tag/middleware/) on [April 3, 2019](https://wakeupandcode.com/middleware-in-asp-net-core/) by [Shahed C](https://wakeupandcode.com/author/admin/).

**Post navigation**

[← Logging in ASP .NET Core](https://wakeupandcode.com/logging-in-asp-net-core/) [.NET Core 3.0, VS2019 and C# 8.0 for ASP .NET Core developers →](https://wakeupandcode.com/net-core-3-vs2019-and-csharp-8/)

**3 thoughts on “Middleware in ASP .NET Core”**

1. Pingback: [Dew Drop – April 4, 2019 (#2932) | Morning Dew](https://www.alvinashcraft.com/2019/04/04/dew-drop-april-4-2019-2932/)
2. Pingback: [The Morning Brew - Chris Alcock » The Morning Brew #2720](http://blog.cwa.me.uk/2019/04/05/the-morning-brew-2720/)
3. Pingback: [Organizational Accounts for ASP .NET Core | Wake Up And Code!](https://wakeupandcode.com/organizational-accounts-for-asp-net-core/)

**Leave a Reply**

This site uses Akismet to reduce spam. [Learn how your comment data is processed](https://akismet.com/privacy/).