## References

https://stackify.com/what-is-kestrel-web-server/

https://stackify.com/kestrel-web-server-asp-net-core-kestrel-vs-iis/

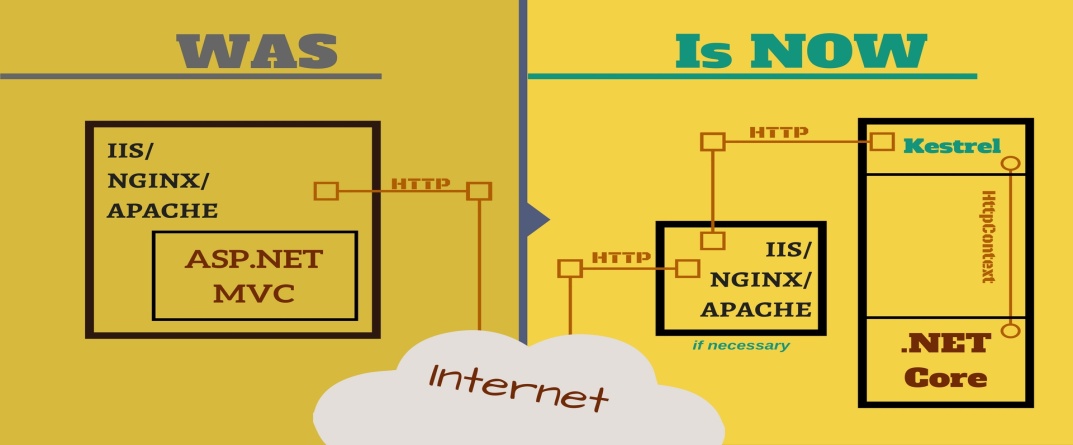
<https://stackify.com/how-to-deploy-asp-net-core-to-iis/>

https://stackoverflow.com/questions/35639205/what-is-kestrel-vs-iis-express

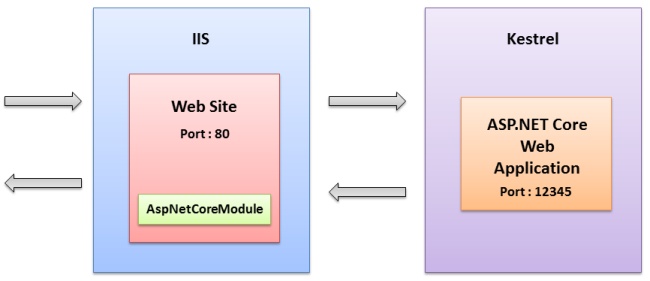
## Kestrel

Kestrel is a webserver included by default as one of fundamentals of ASP.NET Core framework. It provides additional layer in communication between Internet and Application. WIth Kestrel, ASP Core applications doesn't require any other server like IIS, Nginx or Apache, though it's still applicable. And sometimes necessary, because Kestrel doesn't support sharing the same IP and port between multiple processes.

To illustrate the change between Core and MVC, please, take a look at the infographic below:



Now, Kestrel runs a loop in which it listens to HTTP requests (it's called event-driven style of programming) from either Internet or proxy server (where the request can be pre-processed) and forwards them under the wrapping of HttpContext. This additional layer between Internet and application makes it possible to run the latter on any kind of base server. Using Kestrel as an in-process server simplifies startup configuration and makes this process consistent across different platforms.



## What is Kestrel Web Server? How It Works, Benefits, and More

Kestrel is open-source (source code available on [GitHub](https://github.com/aspnet/KestrelHttpServer" \t "_blank)), event-driven, asynchronous I/O based server used to host ASP.NET applications on any platform. It’s a listening server and a command-line interface. You install the listening server on a Windows or Linux server and the command-line interface on your computer.

It was launched by Microsoft along with ASP.NET Core. All ASP.NET Core apps utilize a new MVC framework and the Kestrel web server. These new apps can run on full .NET Framework or .NET Core. ASP.NET is used on [15% of all websites](https://w3techs.com/technologies/details/pl-aspnet/all/all) for which server-side programming languages are known, and as such, it’s important to understand how Kestrel works and the benefits it offers, so that’s why we’ve decided to take a closer look at this web server in today’s post.

**A Definition of the Kestrel Web Server**

Kestrel is considered a preferred web server for newer ASP.NET applications (see this post for a comparison to IIS and [why you need both](https://stackify.com/kestrel-web-server-asp-net-core-kestrel-vs-iis/)). It is [based on the libuv library](http://blog.dotnetnerd.dk/post/2015/11/08/Kestrel-the-new-web-server-for-ASPNET-5.aspx), the same one used by node.js. Libuv supports an event-driven style of programming.

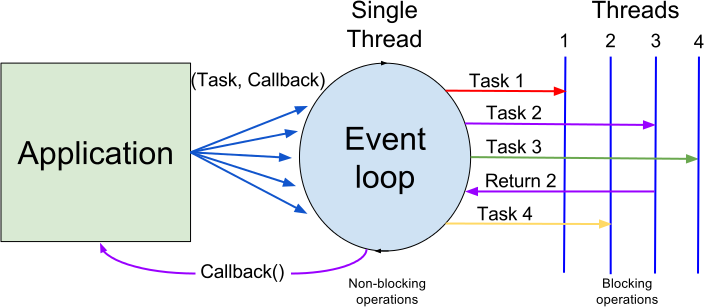
Some of its core utilities include:

* Non-blocking network support
* Asynchronous file system access
* Timers
* Child processes

It allows ASP.NET Core applications to be run easily on other cross-platform webservers such as Nginx and Apache, without the need to address varying startup configurations. By using Kestrel as an in-process server, applications will have a consistent process (Startup (Main(), Startup.ConfigireServices(), Startup.Configure()) ) even with cross-platform support.

**How Kestrel Web Server Works**

Applications are often written to respond to human actions. With event-driven programming, there is a loop that listens for events. It then triggers a callback function. To reduce the number of SYS calls, all other work is executed in managed code on standard .NET worker threads.



Kestrel provides an event loop and callback-based notifications of I/O. Libuv manages the gathering and monitoring of events from the OS. Moreover, the user can register callbacks as an event occurs. So, Kestrel uses libuv for I/O work and supports running multiple event loops.

Because it is so lightweight, Kestrel does not allow you to do SSL termination, URL rewrites, or GZip compression, but that same lightweight design can make other web servers look slow by comparison. It was built to be fast; in fact, it’s six times faster than node.js for static and plain text operations.

**Benefits of Kestrel**

Kestrel has support on all platforms and versions supported by .NET Core. Furthermore, it is included by default in ASP.NET Core new project templates and can provide [better request processing performance](http://www.codedigest.com/quick-start/5/learn-kestrel-webserver-in-10-minutes). When you create a new project in Visual Studio, your project is [automatically configured](http://tostring.it/2016/01/12/Using-Kestrel-with-ASPNET-5/) to run in Kestrel.

As stated earlier, it’s not a fully-featured web server, but that’s precisely why it’s fast. If you feel the need for speed, Kestrel is the answer–especially since it is designed to be used in production for ASP.NET.

What you can do is run it behind a more fully-featured webserver such as IIS or [NGINX](https://aspnetmonsters.com/2016/07/2016-07-17-nginx/). You can run it behind IIS using the HttpPlatformHandler or behind IIS Express using HttpPlatformHandler by Visual Studio. Moreover, you’ll want to support it in your ASP.NET Core projects so that they can be conveniently run by developers on any of the supported platforms.

Now, even if you are not working cross-platform, you can run ASP.NET on a web server straight from the command line.

Since Kestrel is not a fully-featured web server, you should [run it behind IIS](https://stackify.com/kestrel-web-server-asp-net-core-kestrel-vs-iis/) or NGINX. It was designed to make ASP.NET as fast as possible but is limited in its ability to manage security and serve static files. If you’re using ASP.NET Core with Kestrel, you can [take advantage of Prefix](https://docs.stackify.com/v1/docs/prefix-using-with-asp-net-corekestrel) for powerful code profiling.

## How to Deploy ASP.NET Core to IIS & How ASP.NET Core Hosting Works

Previously, I discussed the differences between [Kestrel vs IIS](https://stackify.com/kestrel-web-server-asp-net-core-kestrel-vs-iis/). In this article, we will review how to deploy an ASP.NET Core application to IIS. Deploying an ASP.NET Core app to IIS isn’t complicated, but ASP.NET Core hosting is a little different than ASP.NET.

How to Configure Your ASP.NET Core App For IIS

The first thing you will notice when creating a new ASP.NET Core project is they are actually console applications. Your project now contains a Program.cs file just like a console app would have and it contains the following code:

public class Program

{

public static void Main(string[] args)

{

var host = new WebHostBuilder()

.UseKestrel()

.UseContentRoot(Directory.GetCurrentDirectory())

.UseIISIntegration()

.UseStartup()

.Build();

host.Run();

}

}

What is the WebHostBuilder?

All ASP.NET Core applications require a **WebHost** object that essentially serves as the application and web server. **WebHostBuilder** is used to configure and create the WebHost. You will normally see UseKestrel() and UseIISIntegration() in the WebHostBuilder setup code.

What do these do?

**UseKestrel()** – This registers the IServer interface for Kestrel as the server that will be used to host your application. In the future, there could be other options, including [WebListener](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/weblistener" \t "_blank) which will be Windows only.

**UseIISIntegration()** – This tells ASP.NET that IIS will be working as a reverse proxy in front of Kestrel. This then specifies some settings around which port Kestrel should listen on, forwarding headers, and other details.

*If you are planning to deploy your application to IIS, UseIISIntegration() is required*

What is AspNetCoreModule?

You may have noticed that ASP.NET Core projects create a web.config file. This is only used when deploying your application to IIS. It registers the **AspNetCoreModule** as an HTTP handler.

Default web.config for ASP.NET Core:

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<system.webServer>

<handlers>

<add name="aspNetCore" path="\*" verb="\*" modules="AspNetCoreModule" resourceType="Unspecified"/>

</handlers>

<aspNetCore processPath="%LAUNCHER\_PATH%" arguments="%LAUNCHER\_ARGS%" stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout" forwardWindowsAuthToken="false"/>

</system.webServer>

</configuration>

AspNetCoreModule handles all incoming traffic to IIS and acts as the reverse proxy that knows how to hand the traffic off to your ASP.NET Core application. You can view the source code of it [on GitHub](https://github.com/aspnet/AspNetCoreModule). It also ensures that your web application is running. It is responsible for starting your process up.

Install .NET Core Windows Server Hosting Bundle

**Before you deploy your application, you need to install the .NET Core hosting bundle for IIS. This will install the .NET Core runtime, libraries, and the ASP.NET Core module for IIS.**

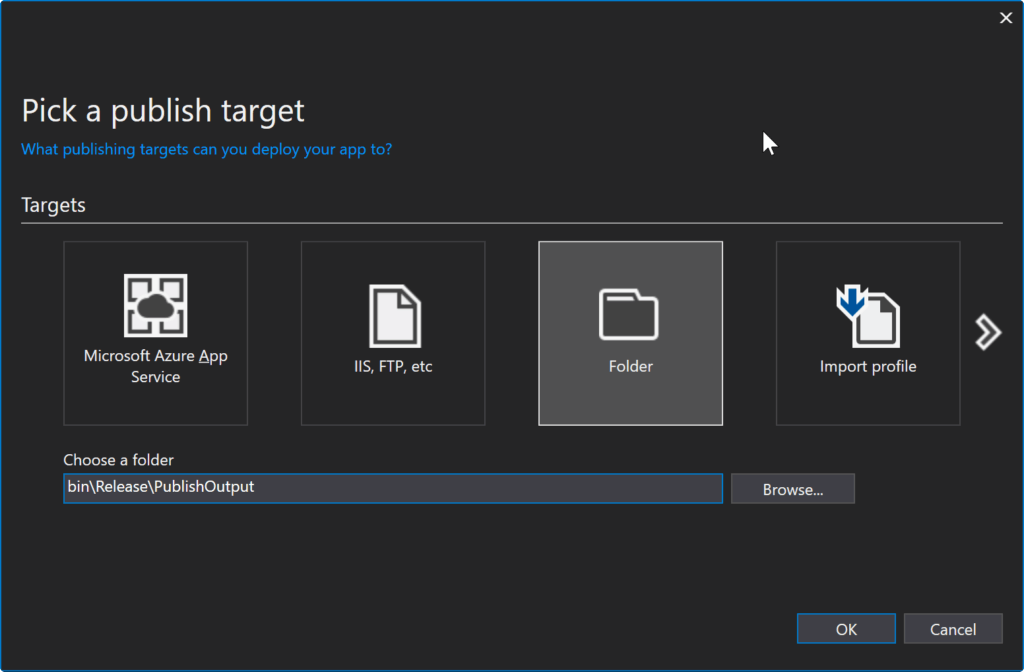
**After installing it, you may need to do a “net stop was /y” and “net start w3svc” to ensure all the changes are picked up for IIS.**

Download: [.NET Core Windows Server Hosting](https://www.microsoft.com/net/download/all) **<- Make sure you pick “Windows Server Hosting!!”**

Steps to Deploy ASP.NET Core to IIS

Before you deploy, you need to make sure that WebHostBuilder is configured properly to use Kestrel and IIS. Your web.config should also exist and look similar to our example above.

Step 1: Publish to a File Folder



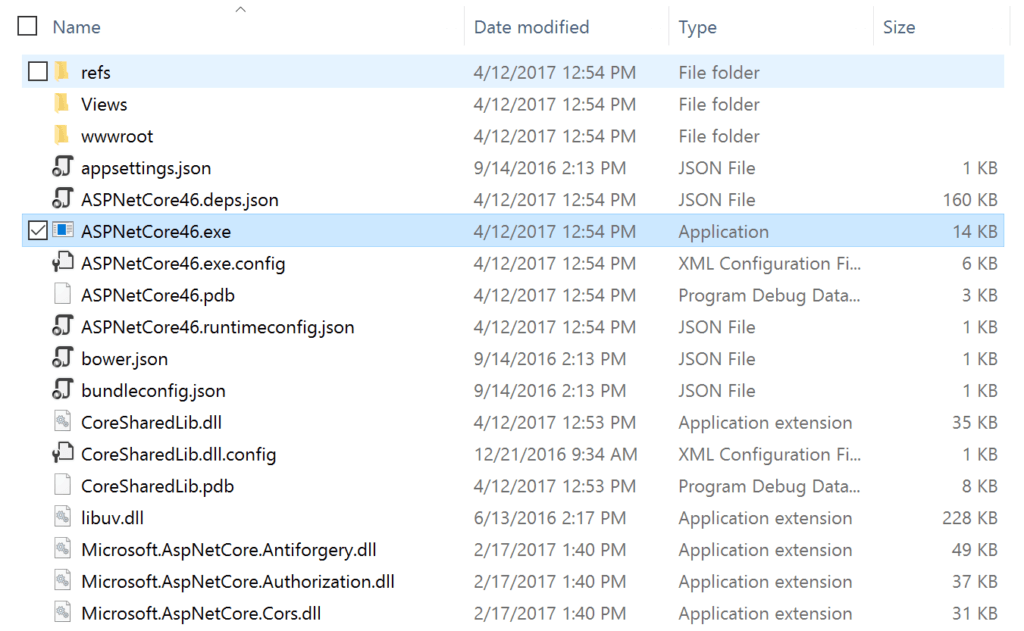
PUBLISH TO FOLDER WITH VISUAL STUDIO 2017

Step 2: Copy Files to Preferred IIS Location

Now you need to copy your publish output to where you want the files to live. If you are deploying to a remote server, you may want to zip up the files and move to the server. If you are deploying to a local dev box, you can copy them locally.

For my example, I am copying the files to C:\inetpub\wwwroot\AspNetCore46

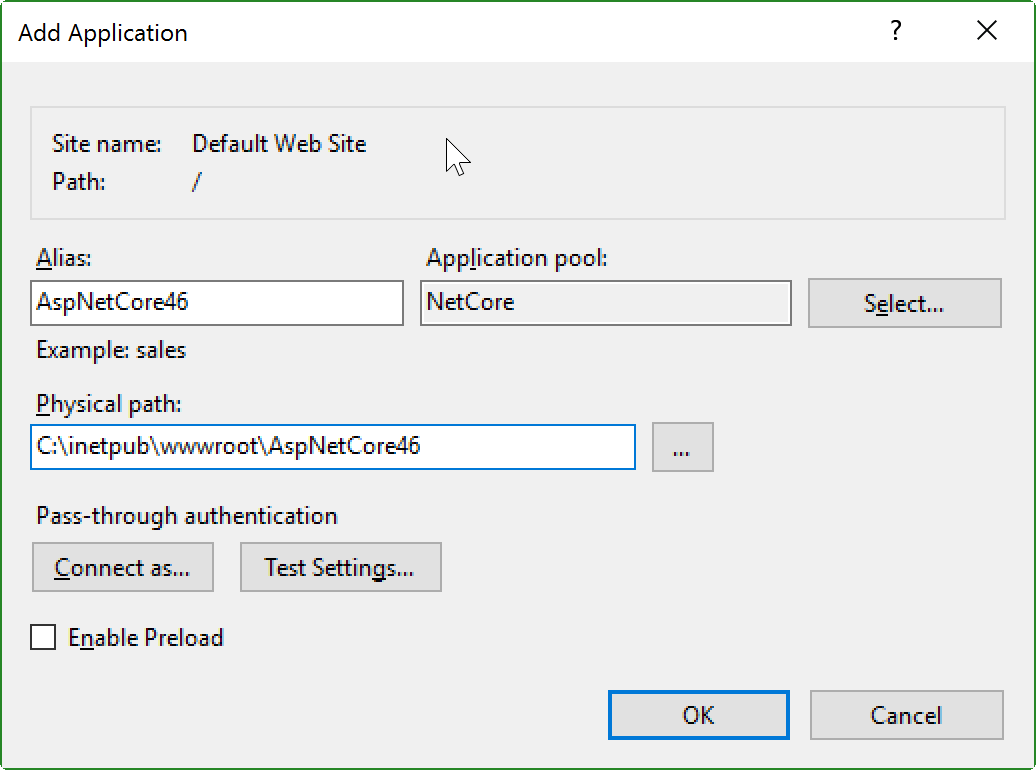
You will notice that with ASP.NET core there is no bin folder and it potentially copies over a ton of different .NET dlls. Your application may also be an EXE file if you are targeting the full .NET Framework. My little sample project had over 100 dlls in the output.



Step 3: Create Application in IIS

First, create a new IIS Application Pool. You will want to create one under the .NET CLR version of “**No Managed Code**“. Since IIS only works as a reverse proxy, it isn’t actually executing any .NET code.

Second, create your new application under your existing IIS Site, or create a new IIS site. Either way, you will want to pick your new IIS Application Pool and point it at the folder you copied your ASP.NET publish output files to.



Step 4: Load Your App!

At this point, your application should load just fine. If it does not, check the output logging from it. Within your web.config file you define how IIS starts up your ASP.NET Core process. Enable output logging by setting **stdoutLogEnabled**=true and you may also want to change the log output location as configured in **stdoutLogFile**. Check out the example web.config before to see where they are set.

Advantages of Using IIS with ASP.NET Core Hosting

Microsoft recommends using IIS with any public facing site for ASP.NET core hosting. IIS provides additional levels of configurability, management, security, logging, and many other things. Check out my blog post about [Kestrel vs IIS](https://stackify.com/kestrel-web-server-asp-net-core-kestrel-vs-iis/) to see a whole matrix of feature differences. It goes more in depth about what Kestrel is and why you need both Kestrel & IIS.

One of the big advantages to using IIS is the **process management**. IIS will automatically **start your app** and potentially restart it if a crash were to occur. If you were running your [ASP.NET Core app as a Windows Service](https://stackify.com/creating-net-core-windows-services/) or console app, you would not have that safety net there to start and monitor the process for you.

I'd like to offer an alternative answer, with some history, so that you might understand why Kestrel comes, even if you only use Windows and IIS.

At the very beginning of ASP.NET development before year 2000, clearly Microsoft created two pieces to host ASP.NET WebForms apps,

Cassini, later became ASP.NET Development Server in Visual Studio. It is a fully managed web server written in C# based on HttpListener. Of course, since it was for development only, many features were never implemented. As Microsoft made the source code of Cassini available for the public, there are third parties who forked the code base and added more features, which started the Cassini family.

ASP.NET support on IIS (revision 1). Because IIS was 4.0 and 5.0/5.1 at that time, which has nothing like application pools, ASP.NET even has its own worker process (aspnet\_wp.exe).

So to develop a web app, you use Cassini, and to deploy you use IIS.

The introduction of application pools in IIS 6 required some changes on ASP.NET side, so aspnet\_wp.exe became obsolete and replaced by aspnet\_isapi.dll. That can be seen as ASP.NET support on IIS revision 2. So ASP.NET apps are being hosted in IIS worker processes w3wp.exe.

The introduction of integrated pipeline in IIS 7 and above required further changes, which replaced aspnet\_isapi.dll with webengine4.dll. That can be seen as ASP.NET support on IIS revision 3. ASP.NET and IIS pipelines are unified.

You can see ASP.NET has become much more complex and tightly integrated with IIS, so Cassini started to show its age, and gradually was replaced by IIS Express (a user mode lite IIS).

Thus, in many cases, when people blame that IIS is slow, they should blame ASP.NET in fact. IIS itself without ASP.NET is pretty fast and stable, while ASP.NET was not developed with enough performance metrics in mind (as WebForms focuses quite a lot of productivities and RAD).

Then in November 2014, ASP.NET 5 (later renamed to ASP.NET Core) was announced and became a cross platform technology. Obviously Microsoft needed a new design to support Windows, macOS, and Linux, where all major web servers, nginx/Apache (or other web servers) should be considered besides IIS.

I think many would agree that Microsoft learned quite a lot from NodeJS, and then designed and developed Kestrel (based on libuv initially but might move to other technology soon). It is a light-weight web server like Cassini initially, but later more features are being added (like another answer commented, much more features so can be treated as a full web server). Though fully managed (some native dependencies exist), it is no longer a toy web server like Cassini.

Then why cannot you just use Kestrel? Why IIS Express and potentially IIS, nginx, or Apache are still needed? That primarily is a result of today's internet practice. Most web sites use reverse proxies to take requests from your web browsers and then forward to the application servers in the background.

IIS Express/IIS/nginx/Apache are the reverse proxy servers

Kestrel/NodeJS/Tomcat and so on are the application servers

Another answer already showed a link to Microsoft documentation, so you can take a look.

Microsoft developed HttpPlatformHandler initially to make IIS a good enough reverse proxy for Java/Python and so on, so planned to use it for ASP.NET Core. Issues started to appear during development, so later Microsoft made ASP.NET Core Module specifically for ASP.NET Core. That's ASP.NET support on IIS revision 4.

Starting from ASP.NET Core 2.2, ASP.NET Core Module for IIS (version 2) can host .NET Core environment inside IIS worker process (w3wp.exe), quite similar to ASP.NET 2.x/4.x. This mode is called ["IIS in-process hosting"](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/aspnet-core-module?view=aspnetcore-2.2#in-process-hosting-model). It can be considered as ASP.NET support on IIS revision 5.

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.1>

ASP.NET Core Middleware

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 is invoked.

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

Request delegates are configured using [Run](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.runextensions.run), [Map](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.mapextensions.map), and [Use](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.useextensions.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.

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. A delegate can also decide to not pass a request to the next delegate, which is called short-circuiting the request pipeline. Short-circuiting is often desirable because it avoids unnecessary work. For example, Static Files Middleware can return a request for a static file and short-circuit the rest of the pipeline. Exception-handling delegates are 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!");

});

}

}

The first [Run](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.runextensions.run) delegate terminates the pipeline.

Chain multiple request delegates together with [Use](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.useextensions.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.");

});

}

}

Warning

Don't call next.Invoke after the response has been sent to the client. Changes to **[HttpResponse](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.http.httpresponse)** after the response has started throw an exception. For example, changes such as 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.

Order

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 middleware components for common app scenarios:

Exception/error handling

HTTP Strict Transport Security Protocol

HTTPS redirection

Static file server

Cookie policy enforcement

Authentication

Session

MVC

C#Copy

public void Configure(IApplicationBuilder app)

{

if (env.IsDevelopment())

{

// When the app runs in the Development environment:

// Use the Developer Exception Page to report app runtime errors.

// Use the Database Error Page to report database runtime errors.

app.UseDeveloperExceptionPage();

app.UseDatabaseErrorPage();

}

else

{

// When the app doesn't run in the Development environment:

// Enable the Exception Handler Middleware to catch exceptions

// thrown in the following middlewares.

// Use the HTTP Strict Transport Security Protocol (HSTS)

// Middleware.

app.UseExceptionHandler("/Error");

app.UseHsts();

}

// Use HTTPS Redirection Middleware to redirect HTTP requests to HTTPS.

app.UseHttpsRedirection();

// Return static files and end the pipeline.

app.UseStaticFiles();

// Use Cookie Policy Middleware to conform to EU General Data

// Protection Regulation (GDPR) regulations.

app.UseCookiePolicy();

// Authenticate before the user accesses secure resources.

app.UseAuthentication();

// If the app uses session state, call Session Middleware after Cookie

// Policy Middleware and before MVC Middleware.

app.UseSession();

// Add MVC to the request pipeline.

app.UseMvc();

}

In the preceding example code, each middleware extension method is exposed on [IApplicationBuilder](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.iapplicationbuilder) through the [Microsoft.AspNetCore.Builder](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder) namespace.

[UseExceptionHandler](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.exceptionhandlerextensions.useexceptionhandler) is the first middleware component added to the pipeline. Therefore, the Exception Handler Middleware catches any exceptions that occur in later calls.

Static Files Middleware is called early in the pipeline so that it can handle requests and short-circuit without going through the remaining components. The Static Files Middleware provides **no**authorization checks. Any files served by it, including those under wwwroot, are publicly available. For an approach to secure static files, see [Static files in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files?view=aspnetcore-2.1).

If the request isn't handled by the Static Files Middleware, it's passed on to the Authentication Middleware ([UseAuthentication](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.authappbuilderextensions.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 Files Middleware before Response Compression Middleware. Static files aren't compressed with this middleware order. The MVC responses from [UseMvcWithDefaultRoute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.mvcapplicationbuilderextensions.usemvcwithdefaultroute) can be compressed.

C#Copy

public void Configure(IApplicationBuilder app)

{

// Static files not compressed by Static Files Middleware.

app.UseStaticFiles();

app.UseResponseCompression();

app.UseMvcWithDefaultRoute();

}

Use, Run, and Map

Configure the HTTP pipeline using Use, Run, and Map. The Use method can short-circuit the pipeline (that is, if it doesn't call a next request delegate). Run is a convention, and some middleware components may expose Run[Middleware] methods that run at the end of the pipeline.

[Map](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.mapextensions.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 segment(s) are removed from HttpRequest.Path and appended to HttpRequest.PathBase for each request.

[MapWhen](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.mapwhenextensions) 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 |
| --- | --- |
| localhost:1234 | Hello from non-Map delegate. |
| localhost:1234/?branch=master | Branch used = master |

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.");

});

}

}

Built-in middleware

ASP.NET Core ships with the following middleware components. The Order column provides notes on the middleware's placement in the request pipeline and under what conditions the middleware may terminate the request and prevent other middleware from processing a request.

| Middleware | Description | Order |
| --- | --- | --- |
| [Authentication](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-2.1) | Provides authentication support. | Before HttpContext.User is needed. Terminal for OAuth callbacks. |
| [Cookie Policy](https://docs.microsoft.com/en-us/aspnet/core/security/gdpr?view=aspnetcore-2.1) | 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](https://docs.microsoft.com/en-us/aspnet/core/security/cors?view=aspnetcore-2.1) | Configures Cross-Origin Resource Sharing. | Before components that use CORS. |
| [Diagnostics](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/error-handling?view=aspnetcore-2.1) | Configures diagnostics. | Before components that generate errors. |
| [Forwarded Headers](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.forwardedheadersextensions) | Forwards proxied headers onto the current request. | Before components that consume the updated fields. Examples: scheme, host, client IP, method. |
| [HTTP Method Override](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.httpmethodoverrideextensions) | Allows an incoming POST request to override the method. | Before components that consume the updated method. |
| [HTTPS Redirection](https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-2.1#require-https) | Redirect all HTTP requests to HTTPS (ASP.NET Core 2.1 or later). | Before components that consume the URL. |
| [HTTP Strict Transport Security (HSTS)](https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-2.1#http-strict-transport-security-protocol-hsts) | Security enhancement middleware that adds a special response header (ASP.NET Core 2.1 or later). | Before responses are sent and after components that modify requests. Examples: Forwarded Headers, URL Rewriting. |
| [MVC](https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-2.1) | Processes requests with MVC/Razor Pages (ASP.NET Core 2.0 or later). | Terminal if a request matches a route. |
| [OWIN](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/owin?view=aspnetcore-2.1) | Interop with OWIN-based apps, servers, and middleware. | Terminal if the OWIN Middleware fully processes the request. |
| [Response Caching](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/middleware?view=aspnetcore-2.1) | Provides support for caching responses. | Before components that require caching. |
| [Response Compression](https://docs.microsoft.com/en-us/aspnet/core/performance/response-compression?view=aspnetcore-2.1) | Provides support for compressing responses. | Before components that require compression. |
| [Request Localization](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization?view=aspnetcore-2.1) | Provides localization support. | Before localization sensitive components. |
| [Routing](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-2.1) | Defines and constrains request routes. | Terminal for matching routes. |
| [Session](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-2.1) | Provides support for managing user sessions. | Before components that require Session. |
| [Static Files](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files?view=aspnetcore-2.1) | Provides support for serving static files and directory browsing. | Terminal if a request matches a file. |
| [URL Rewriting](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/url-rewriting?view=aspnetcore-2.1) | Provides support for rewriting URLs and redirecting requests. | Before components that consume the URL. |
| [WebSockets](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/websockets?view=aspnetcore-2.1) | Enables the WebSockets protocol. | Before components that are required to accept WebSocket requests. |

Write middleware

Middleware is generally encapsulated in a class and exposed with an extension method. Consider the following middleware, which sets the culture for the current request from a query string:

C#Copy

public class Startup

{

public void Configure(IApplicationBuilder app)

{

app.Use((context, next) =>

{

var cultureQuery = context.Request.Query["culture"];

if (!string.IsNullOrWhiteSpace(cultureQuery))

{

var culture = new CultureInfo(cultureQuery);

CultureInfo.CurrentCulture = culture;

CultureInfo.CurrentUICulture = culture;

}

// Call the next delegate/middleware in the pipeline

return next();

});

app.Run(async (context) =>

{

await context.Response.WriteAsync(

$"Hello {CultureInfo.CurrentCulture.DisplayName}");

});

}

}

The preceding sample code is used to demonstrate creating a middleware component. For ASP.NET Core's built-in localization support, see [Globalization and localization in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization?view=aspnetcore-2.1).

You can test the middleware by passing in the culture, for example http://localhost:7997/?culture=no.

The following code moves the middleware delegate to a class:

C#Copy

using Microsoft.AspNetCore.Http;

using System.Globalization;

using System.Threading.Tasks;

namespace Culture

{

public class RequestCultureMiddleware

{

private readonly RequestDelegate \_next;

public RequestCultureMiddleware(RequestDelegate next)

{

\_next = next;

}

public async Task InvokeAsync(HttpContext context)

{

var cultureQuery = context.Request.Query["culture"];

if (!string.IsNullOrWhiteSpace(cultureQuery))

{

var culture = new CultureInfo(cultureQuery);

CultureInfo.CurrentCulture = culture;

CultureInfo.CurrentUICulture = culture;

}

// Call the next delegate/middleware in the pipeline

await \_next(context);

}

}

}

The following extension method exposes the middleware through [IApplicationBuilder](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.iapplicationbuilder):

C#Copy

using Microsoft.AspNetCore.Builder;

namespace Culture

{

public static class RequestCultureMiddlewareExtensions

{

public static IApplicationBuilder UseRequestCulture(

this IApplicationBuilder builder)

{

return builder.UseMiddleware<RequestCultureMiddleware>();

}

}

}

The following code calls the middleware from Startup.Configure:

C#Copy

public class Startup

{

public void Configure(IApplicationBuilder app)

{

app.UseRequestCulture();

app.Run(async (context) =>

{

await context.Response.WriteAsync(

$"Hello {CultureInfo.CurrentCulture.DisplayName}");

});

}

}

Middleware should follow the [Explicit Dependencies Principle](https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles#explicit-dependencies) by exposing its dependencies in its constructor. Middleware is constructed once per application lifetime. See the [Per-request dependencies](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.1#per-request-dependencies) section if you need to share services with middleware within a request.

Middleware components can resolve their dependencies from [dependency injection (DI)](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.1) through constructor parameters. [UseMiddleware<T>](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.usemiddlewareextensions.usemiddleware" \l "Microsoft_AspNetCore_Builder_UseMiddlewareExtensions_UseMiddleware_Microsoft_AspNetCore_Builder_IApplicationBuilder_System_Type_System_Object___) can also accept additional parameters directly.

Per-request dependencies

Because middleware is constructed at app startup, not per-request, scoped lifetime services used by middleware constructors aren't shared with other dependency-injected types during each request. If you must share a scoped service between your middleware and other types, add these services to the Invoke method's signature. The Invoke method can accept additional parameters that are populated by DI:

C#Copy

public class CustomMiddleware

{

private readonly RequestDelegate \_next;

public CustomMiddleware(RequestDelegate next)

{

\_next = next;

}

// IMyScopedService is injected into Invoke

public async Task Invoke(HttpContext httpContext, IMyScopedService svc)

{

svc.MyProperty = 1000;

await \_next(httpContext);

}

}

Additional resources

[Migrate HTTP handlers and modules to ASP.NET Core middleware](https://docs.microsoft.com/en-us/aspnet/core/migration/http-modules?view=aspnetcore-2.1)

[Application startup in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/startup?view=aspnetcore-2.1)

[Request Features in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/request-features?view=aspnetcore-2.1)

[Factory-based middleware activation in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/extensibility?view=aspnetcore-2.1)

[Middleware activation with a third-party container in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/extensibility-third-party-container?view=aspnetcore-2.1)

## Hosting models

### In-process hosting model

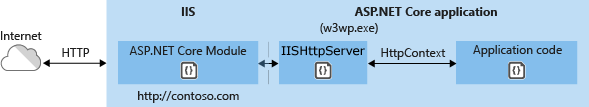
Using in-process hosting, an ASP.NET Core app runs in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter, a network interface that returns outgoing network traffic back to the same machine. IIS handles process management with the [Windows Process Activation Service (WAS)](https://docs.microsoft.com/en-us/iis/manage/provisioning-and-managing-iis/features-of-the-windows-process-activation-service-was).

The [ASP.NET Core Module](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/aspnet-core-module?view=aspnetcore-3.1):

* Performs app initialization.
  + Loads the [CoreCLR](https://docs.microsoft.com/en-us/dotnet/standard/glossary" \l "coreclr).
  + Calls Program.Main.
* Handles the lifetime of the IIS native request.

The in-process hosting model isn't supported for ASP.NET Core apps that target the .NET Framework.

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted in-process:



A request arrives from the web to the kernel-mode HTTP.sys driver. The driver routes the native request to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The ASP.NET Core Module receives the native request and passes it to IIS HTTP Server (IISHttpServer). IIS HTTP Server is an in-process server implementation for IIS that converts the request from native to managed.

After the IIS HTTP Server processes the request, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an HttpContext instance to the app's logic. The app's response is passed back to IIS through IIS HTTP Server. IIS sends the response to the client that initiated the request.

In-process hosting is opt-in for existing apps, but [dotnet new](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-new) templates default to the in-process hosting model for all IIS and IIS Express scenarios.

CreateDefaultBuilder adds an [IServer](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.hosting.server.iserver) instance by calling the [UseIIS](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.hosting.webhostbuilderiisextensions.useiis) method to boot the [CoreCLR](https://docs.microsoft.com/en-us/dotnet/standard/glossary" \l "coreclr) and host the app inside of the IIS worker process (w3wp.exe or iisexpress.exe). Performance tests indicate that hosting a .NET Core app in-process delivers significantly higher request throughput compared to hosting the app out-of-process and proxying requests to [Kestrel](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-3.1) server.

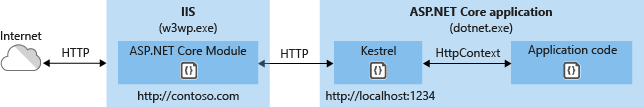
**Note**

Apps published as a single file executable can't be loaded by the in-process hosting model.

### Out-of-process hosting model

Because ASP.NET Core apps run in a process separate from the IIS worker process, the ASP.NET Core Module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the [Windows Process Activation Service (WAS)](https://docs.microsoft.com/en-us/iis/manage/provisioning-and-managing-iis/features-of-the-windows-process-activation-service-was).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted out-of-process:



Requests arrive from the web to the kernel-mode HTTP.sys driver. The driver routes the requests to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The module forwards the requests to Kestrel on a random port for the app, which isn't port 80 or 443.

The module specifies the port via an environment variable at startup, and the [UseIISIntegration](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.hosting.webhostbuilderiisextensions.useiisintegration) extension configures the server to listen on http://localhost:{PORT}. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding, so requests are forwarded over HTTP even if received by IIS over HTTPS.

After Kestrel picks up the request from the module, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an HttpContext instance to the app's logic. Middleware added by IIS Integration updates the scheme, remote IP, and pathbase to account for forwarding the request to Kestrel. The app's response is passed back to IIS, which pushes it back out to the HTTP client that initiated the request.