Contents

[Middleware 2](#_Toc53386859)

[What are Middleware Components? 2](#_Toc53386860)

[Create a middleware pipeline with IApplicationBuilder 3](#_Toc53386861)

[Example of built in middleware for routing 5](#_Toc53386862)

[Endpoint 5](#_Toc53386863)

[UseAuthentication and UseAuthorization Middleware 6](#_Toc53386864)

[app.Use vs app.Run in ASP.NET Core middleware 6](#_Toc53386865)

[Build – in middleware 6](#_Toc53386866)

[About IApplicationBuilder Interface 6](#_Toc53386867)

[How to create custom Middleware? 7](#_Toc53386868)

[Step – 1 7](#_Toc53386869)

[Extension method to add (expose) custom middleware 9](#_Toc53386870)

[Call custom middleware in start up 9](#_Toc53386871)

[References: 10](#_Toc53386872)

# Middleware

## What are Middleware Components?

Middleware are software components that are assembled into an application pipeline to handle requests and responses. Each component chooses whether to pass the request on to the next component in the pipeline, and can perform certain actions before and after the next component is invoked in the pipeline.

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

Pieces of code which are added to your application’s pipeline. Each piece of code has opportunity to perform some logic and handle the request.

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 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.

## 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.

|  |
| --- |
| 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.

|  |
| --- |
| 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.

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.

## Example of built in middleware for routing

Routing uses a pair of middleware, registered by UseRouting and UseEndpoints:

|  |  |
| --- | --- |
| Middleware | Description |
| app.UseRouting(); | UseRouting adds route matching to the middleware pipeline. This middleware looks at the set of endpoints defined in the app, and selects the best match based on the request. |
| app.UseEndpoints() | UseEndpoints adds endpoint execution to the middleware pipeline. It runs the delegate associated with the selected endpoint.  app.UseEndpoints(endpoints =>  {  endpoints.MapGet("/", async context =>  {  await context.Response.WriteAsync("Hello World!");  });  });  The preceding example includes a single route to code endpoint using the MapGet method:   * When an HTTP GET request is sent to the root URL /: * The request delegate shown executes. * Hello World! is written to the HTTP response. By default, the root URL / is https://localhost:5001/. * If the request method is not GET or the root URL is not /, no route matches and an HTTP 404 is returned. |

## Endpoint

The MapGet method is used to define an endpoint. An endpoint is something that can be:

* Selected, by matching the URL and HTTP method.
* Executed, by running the delegate.

Endpoints that can be matched and executed by the app are configured in UseEndpoints.

For example, MapGet, MapPost, and similar methods connect request delegates to the routing system. Additional methods can be used to connect ASP.NET Core framework features to the routing system:

* MapRazorPages for Razor Pages
* MapControllers for controllers
* MapHub<THub> for SignalR
* MapGrpcService<TService> for gRPC

## UseAuthentication and UseAuthorization Middleware

Calling UseAuthentication and UseAuthorization adds the authentication and authorization middleware. These middleware are placed between UseRouting and UseEndpoints so that they can:

* See which endpoint was selected by UseRouting.
* Apply an authorization policy before UseEndpoints dispatches to the endpoint.

## app.Use vs app.Run in ASP.NET Core middleware

Middleware are executed in the same order in which they are added. The difference is, middleware defined using app.Use may call next middleware component in the pipeline. On the other hand, middlware defined using app.Run will never call subsequent middleware.

Run method adds a RequestDelegate which is terminal to the pipeline. Run() is an extension method on IApplicationBuilder instance which adds a terminal middleware to the application's request pipeline.

## Build – in middleware

|  |  |
| --- | --- |
| Middleware | Description |
| Authentication | Provides authentication support |
| Authorization | Provides authorization support |
| CORS | Configures Cross-Origin Resource Sharing |
| Static Files | Provides support for serving static files and directory browsing |

## About IApplicationBuilder Interface

Defines a class that provides the mechanisms to configure an application's request pipeline.

## How to create custom Middleware?

Middleware is generally encapsulated in a class and exposed with an extension method.

The middleware class must include:

* A public constructor with a parameter of type RequestDelegate.
* A public method named Invoke or InvokeAsync. This method must:
  + Return a Task.
  + Accept a first parameter of type HttpContext.

Additional parameters for the constructor and Invoke/InvokeAsync are populated by dependency injection (DI).

### Step – 1

Create a class name MyLoggerMiddleware

|  |
| --- |
| public class MyLoggerMiddleware  {  private readonly RequestDelegate \_next;  private readonly ILoggerFactory \_loggerFactory;  public MyLoggerMiddleware(RequestDelegate next, ILoggerFactory loggerFactory)  {  \_next = next;  \_loggerFactory = loggerFactory;  }  public async Task InvokeAsync(HttpContext context)  {  var logger = \_loggerFactory.CreateLogger<MyLoggerMiddleware>();  try  {  logger.LogInformation("Performing Middleware Operation - Entering Middleware ");  logger.LogInformation("Context.Request.Host.Value " + context.Request.Host.Value);  logger.LogInformation("Context.Request.Path.Value " + context.Request.Path.Value);  // await context.Response.WriteAsync( "Short Circuit!");  await \_next(context);  logger.LogInformation("Performing Middleware Operation - Exit Middleware ");  }  catch (Exception ex)  {  logger.LogError($"Something went wrong: {ex.Message}");  }  }  } |

Reference to above code snippet

private readonly RequestDelegate \_next;

private readonly ILoggerFactory \_loggerFactory;

public MyLoggerMiddleware(RequestDelegate next, ILoggerFactory loggerFactory)

{

\_next = next;

\_loggerFactory = loggerFactory;

}

\_next initialized with object of RequestDelegate

\_loggerFactory initialized with object of ILoggerFactory

public async Task InvokeAsync(HttpContext context)

{

var logger = \_loggerFactory.CreateLogger<MyLoggerMiddleware>();

try

{

logger.LogInformation("Performing Middleware Operation - Entering Middleware ");

logger.LogInformation("Context.Request.Host.Value " + context.Request.Host.Value);

logger.LogInformation("Context.Request.Path.Value " + context.Request.Path.Value);

// await context.Response.WriteAsync( "Short Circuit!");

await \_next(context);

logger.LogInformation("Performing Middleware Operation - Exit Middleware ");

}

catch (Exception ex)

{

logger.LogError($"Something went wrong: {ex.Message}");

}

}

InvokeAsync method implemented with HttpContext as an argument. Context allows you to interact (get/set) with Request and Response object.

Example

|  |
| --- |
| context.Request.Host.Value allows you to access host name of from request object such as localhost:44341 |
| context.Response.WriteAsync used to send context as a response. |

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.

Example to call next middleware in pipeline into InvokeAsync method

// Call the next delegate/middleware in the pipeline

await \_next(context);

If you do not call this method it prevents further middleware from processing the request. Termed as short-circuits.

### Extension method to add (expose) custom middleware

The following extension method exposes the middleware through IApplicationBuilder:

|  |
| --- |
| public static class MyLoggerMiddlewareExtension  {  public static IApplicationBuilder UseMyLoggerMiddleware(this IApplicationBuilder app)  {  return app.UseMiddleware<MyLoggerMiddleware>();  }  } |

### Call custom middleware in start up

The following code calls the middleware from Startup.Configure:

|  |
| --- |
| public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  }  //Added Custom Middleware to request pipeline using custome extension method - UseMyLoggerMiddleware  app.UseMyLoggerMiddleware();  app.UseRouting();  app.Run(context =>  {  context.Response.WriteAsync("We will do it ");  return Task.CompletedTask;  });  } |

## References:

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1>

<https://www.thecodebuzz.com/logging-middleware-using-loggerfactory-asp-net-core/>

Found that method of custom middleware called twice. Logger allows me to determine this behavior. Reference to understand this:

<https://gaunacode.com/aspnetcoremiddleware>

<https://stackoverflow.com/questions/17372743/django-custom-middleware-called-twice>

All your middleware get called twice: 1 time for the page you have requested + 1 time for the favicon (which is always called by the browser). The solution is simple: don't use django urls to serve your vaficon, instead put something like:

<https://aspnetcore.readthedocs.io/en/stable/fundamentals/middleware.html>